Computer Physics Communications 240 (2019) 120–137

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

A GPU compatible quasi-Monte Carlo integrator interfaced to pySecDec ☆

S. Borowka [a], G. Heinrich [b,*], S. Jahn [b], S.P. Jones [a,b], M. Kerner [b,c], J. Schlenk [d]

[a] Theoretical Physics Department, CERN, Geneva, Switzerland
[b] Max Planck Institute for Physics, Föhringer Ring 6, 80805 München, Germany
[c] Physik-Institut, Universität Zürich, Winterthurerstrasse 190, 8057 Zürich, Switzerland
[d] Institute for Particle Physics Phenomenology, University of Durham, Durham DH1 3LE, UK

ARTICLE INFO

Article history:
Received 22 December 2018
Received in revised form 20 February 2019
Accepted 27 February 2019
Available online 7 March 2019

Keywords:
Perturbation theory
Feynman diagrams
Multi-loop
Numerical integration

ABSTRACT

The purely numerical evaluation of multi-loop integrals and amplitudes can be a viable alternative to analytic approaches, in particular in the presence of several mass scales, provided sufficient accuracy can be achieved in an acceptable amount of time. For many multi-loop integrals, the fraction of time required to perform the numerical integration is significant and it is therefore beneficial to have efficient and well-implemented numerical integration methods. With this goal in mind, we present a new stand-alone integrator based on the use of (quasi-Monte Carlo) rank-1 shifted lattice rules. For integrals with high variance we also implement a variance reduction algorithm based on fitting a smooth function to the inverse cumulative distribution function of the integrand dimension-by-dimension.

Additionally, the new integrator is interfaced to pySecDec to allow the straightforward evaluation of multi-loop integrals and dimensionally regulated parameter integrals. In order to make use of recent advances in parallel computing hardware, our integrator can be used both on CPUs and CUDA compatible GPUs where available.

Program summary

Program Title: pySecDec, qmc
Program Files doi: `http://dx.doi.org/10.17632/dnrkf5jxzh.2`
Licensing provisions: GNU General Public License v3
Programming language: python, FORM, C++, CUDA
External routines/libraries: catch [1], gsl [2], numpy [3], sympy [4], Nauty [5], Cuba [6], FORM [7], Normaliz [8]. The program can also be used in a mode which does not require Normaliz.
Journal reference of previous version: Comput. Phys. Commun. 222 (2018) 313–326.
Does the new version supersede the previous version?: Yes
Nature of problem: Extraction of ultraviolet and infrared singularities from parametric integrals appearing in higher order perturbative calculations in quantum field theory. Numerical integration in the presence of integrable singularities (e.g. kinematic thresholds).
Solution method: Algebraic extraction of singularities within dimensional regularization using iterated sector decomposition. This leads to a Laurent series in the dimensional regularization parameter ϵ (and optionally other regulators), where the coefficients are finite integrals over the unit-hypercube. Those integrals are evaluated numerically by Monte Carlo integration. The integrable singularities are handled by choosing a suitable integration contour in the complex plane, in an automated way. The parameter integrals forming the coefficients of the Laurent series in the regulator(s) are provided in the form of libraries which can be linked to the calculation of (multi-) loop amplitudes.
Restrictions: Depending on the complexity of the problem, limited by memory and CPU/GPU time.
References:
[1] https://github.com/philsquared/Catch/.
[2] http://www.gnu.org/software/gsl/.
[3] http://www.numpy.org/.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655)..

* Corresponding author.
E-mail address: gudrun@mpp.mpg.de (G. Heinrich).

https://doi.org/10.1016/j.cpc.2019.02.015

[4] http://www.sympy.org/.
[5] http://pallini.di.uniroma1.it/.
[6] T. Hahn, "CUBA: A Library for multidimensional numerical integration," Comput. Phys. Commun. 168 (2005) 78 [hep-ph/0404043], http://www.feynarts.de/cuba/.
[7] J. Kuipers, T. Ueda and J. A. M. Vermaseren, "Code Optimization in FORM," Comput. Phys. Commun. 189 (2015) 1 [arXiv:1310.7007], http://www.nikhef.nl/~form/.
[8] W. Bruns, B. Ichim, B. and T. Römer, C. Söger, "Normaliz. Algorithms for rational cones and affine monoids." http://www.math.uos.de/normaliz/.

1. Introduction

High energy particle physics is in an era where the current underlying theory, the Standard Model (SM), is very well tested experimentally, as well as consistent and therefore predictive from a theoretical point of view. This means that we can control the SM predictions very well, and so should be able – at least in principle – to identify physics beyond the SM even if it is showing up only in small deviations.

In practice, there are several obstacles when trying to increase the precision of theoretical predictions. Focusing on problems accessible to perturbation theory, a major obstacle is the fast increase in complexity of the calculation as the number of loops and the number of kinematic scales increases. Despite the remarkable progress that has been achieved in the analytic calculation of multi-loop amplitudes and integrals in the last few years, analytical approaches are only at the beginning of a journey into largely unexplored mathematical territory if the function class of the results goes beyond multiple polylogarithms (MPLs), typically involving elliptic or hyper-elliptic functions, see e.g. [1–13].

On the other hand, (semi-)numerical approaches do not necessarily become less efficient if the result leaves the class of MPLs. This is one of the reasons why it is important to develop numerical methods which are fast and accurate enough to provide results where analytic approaches are at their limits. Sector decomposition [14–17] is an example of such a method; other recent semi-numerical methods are described e.g. in Refs. [18–26]. Sector decomposition is a procedure which can be applied to dimensionally regulated integrals in order to factorize singularities in the regulator. The resulting finite parameter integrals, which form the coefficients at each order in the regulator, can then be numerically integrated. There are several public implementations of sector decomposition [27–35]. Recently, an analytical method, based on sector decomposition followed by a series expansion in the Feynman parameters and analytic integration, has been worked out in Ref. [36].

Currently, in the publicly available sector decomposition tools, numerical integration mostly relies on either deterministic integration rules for integrals of low dimensionality or Monte Carlo integration, as implemented in the Cuba library [37,38]. However, the integration error for Monte Carlo integration scales only like $1/\sqrt{n}$, where n is the number of samples, which limits the accuracy that can be obtained in a given integration time. To improve on this, a different integration method has to be chosen. One such method is the *quasi-Monte Carlo* (QMC) method [39], where the integration error scales like $1/n$ or better, rather than $1/\sqrt{n}$. In order for this scaling to be achieved, the integrand functions need to fulfil certain requirements. The QMC method discussed herein was first applied to functions produced by the sector decomposition algorithm in Ref. [40], where it was shown practically that the conditions for $1/n$ or better scaling are usually met and the good performance of Graphics Processing Units (GPUs) when evaluating such functions was also demonstrated. An application of quasi-Monte Carlo methods to two- and three-loop integrals also has been presented in Ref. [41]. The QMC method, implemented to run on GPUs, has already been applied successfully to phenomenological applications involving multi-scale two-loop integrals including Higgs–boson pair production [42,43] and H+jet production [44] at NLO.

In this work, we present a new stand-alone QMC integrator capable of utilizing multiple cores of Central Processing Units (CPUs) and multiple Graphics Processing Units (GPUs). We also present a new version of the program pySecDec which makes available our QMC implementation as an additional integrator. Furthermore, we present and implement a method for combining the QMC integration with importance sampling. We emphasize that our QMC implementation can also be straightforwardly used outside of the pySecDec context.

The outline of the paper is as follows. In Section 2 we give an overview on the QMC method as implemented in our program and describe our variance reduction procedure. Section 3 is dedicated to the stand-alone usage of the QMC integrator library, we also describe the design of the library and some basics regarding the use of GPUs. In Section 4 we explain the usage of the QMC integrator within pySecDec and describe various examples. Section 5 is dedicated to profiling the QMC method and our implementation. After we conclude in Section 6, we provide detailed API documentation in Appendix.

2. Description of the QMC method

2.1. Quasi-Monte Carlo integration

Our aim is to numerically compute the multiple integral of a function $f : \mathbb{R}^d \to \mathbb{R}$ or $f : \mathbb{R}^d \to \mathbb{C}$ over a d-dimensional unit hypercube $[0, 1]^d$,

$$I[f] \equiv \int_{[0,1]^d} \mathrm{d}\mathbf{x}\, f(\mathbf{x}) \equiv \int_0^1 \mathrm{d}x_1 \cdots \mathrm{d}x_d\, f(x_1, \ldots, x_d)\,. \tag{1}$$

In this section we will briefly introduce the concept of quasi-Monte Carlo integration and state the most relevant results and formulae. The study of QMC integration has produced a vast amount of literature, for a more thorough review we refer the reader to the existing mathematical literature, for example Refs. [39,45] and references therein.

Unlike Monte Carlo integrators, quasi-Monte Carlo (QMC) integrators are based on a predominantly deterministic numerical integration. An unbiased estimate $\bar{Q}_{n,m}[f]$ of the integral $I[f]$ can be obtained from the following (QMC) cubature rule, known as a rank-1 shifted lattice (R1SL) rule [39]:

$$I[f] \approx \bar{Q}_{n,m}[f] \equiv \frac{1}{m}\sum_{k=0}^{m-1} Q_n^{(k)}[f],$$
$$Q_n^{(k)}[f] \equiv \frac{1}{n}\sum_{i=0}^{n-1} f\left(\left\{\frac{i\mathbf{z}}{n} + \boldsymbol{\Delta}_k\right\}\right)\,. \tag{2}$$

The rank of the rule denotes the minimal number of generating vectors required to generate the lattice rule. In this work we will

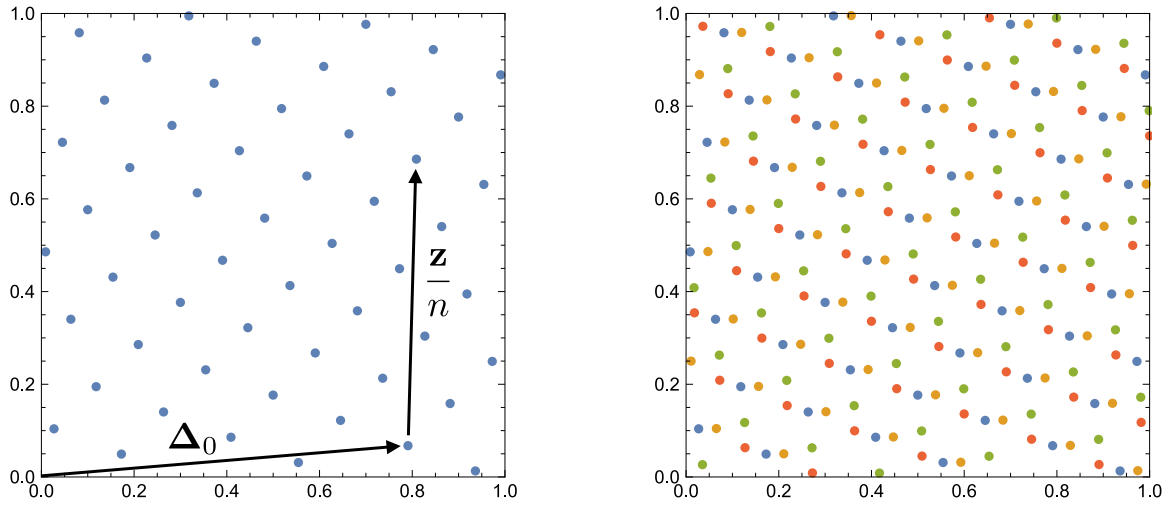

Fig. 1. (Left panel) A $d = 2$ dimensional R1SL with $n = 55$ points, generating vector $\mathbf{z} = (1, 34)$ and random shift $\boldsymbol{\Delta}_0$. (Right panel) A R1SL produced with three additional random shifts, which can be used to estimate the mean-square error as described in the text.

consider only rank-1 lattices i.e. those generated by a single generating vector. The estimate of the integral depends on the number of lattice points n and the number of random shifts m. The shift vectors $\boldsymbol{\Delta}_k \in [0, 1)^d$ are d-dimensional vectors with components consisting of independent, uniformly distributed random real numbers in the interval $[0, 1)$. The generating vector $\mathbf{z} \in \mathbb{Z}^d$ is a fixed d-dimensional vector of integers coprime to n. The curly brackets indicate that the fractional part of each component is taken, such that all arguments of f remain in the interval $[0, 1)$.

A reliable estimate of the integral can be obtained even without random shifts provided that the lattice is sufficiently large, however, the random shifts allow the remaining error to be estimated. More precisely, an unbiased estimate of the mean-square error can be obtained from the random shifts of the lattice according to

$$\sigma^2_{n,m}[f] \equiv \mathrm{Var}[\bar{Q}_{n,m}[f]] \approx \frac{1}{m(m-1)} \sum_{k=0}^{m-1} (Q_n^{(k)}[f] - \bar{Q}_{n,m}[f])^2 \,. \quad (3)$$

In typical applications only 10–20 random shifts are required to obtain a reliable estimate of the error.

In Fig. 1 an example shifted lattice is shown. In the left panel a single lattice is displayed. The zeroth point is shifted from the origin by the random shift vector $\boldsymbol{\Delta}_0$. Further points are generated by adding $\mathbf{z}/n$ and wrapping back into the unit square as necessary. The lattice displayed contains a total of $n = 55$ points. In the right panel, three additional shifted lattices are displayed. They are generated by shifting the original lattice and can be used to produce an estimate of the integration error using Eq. (3).

The classical theoretical error bounds on QMC rules take the form of a product

$$|I[f] - Q_n[f]| \leq D(\mathbf{t}_0, \ldots, \mathbf{t}_{n-1}) V[f], \quad (4)$$

where $\mathbf{t}_i$ are the cubature points generated by the generating vector(s), D is the discrepancy of the point set and V is the variation of f. The discrepancy depends only on the points and the variation depends only on the integrand. If f can be differentiated once with respect to each variable then it can be proven that for a particular choice of cubature points (or, equivalently, a particular generating vector) QMC methods converge as $\mathcal{O}((\log n)^d/n)$. This error bound grows exponentially with dimension, seemingly implying that QMC integration is not useful in a large number of dimensions.

However, by working with weighted function spaces, it can be shown that the error bound can be independent of the dimension provided that the variables of the integrand f have some varying degree of importance. In the modern literature, error bounds have been studied in terms of the product

$$|I[f] - Q_n[f]| \leq e_{\boldsymbol{\gamma}}(\mathbf{t}_0, \ldots, \mathbf{t}_{n-1}) \|f\|_{\boldsymbol{\gamma}}, \quad (5)$$

where $e_{\boldsymbol{\gamma}}$ is the worst case error in a weighted function space with weights $\boldsymbol{\gamma}$ and $\|f\|_{\boldsymbol{\gamma}}$ is the norm of f in the weighted space.

Following Ref. [39] we will discuss two function spaces (Sobolev spaces and Korobov spaces) that allow important properties of the QMC to be proven. In both cases we will state theorems from the literature that bound the worst case error for a rank-1 shifted lattice rule in the corresponding function space. By definition, the worst case error for a shifted lattice rule in the weighted function space is the largest possible error for any function with norm less than or equal to 1,

$$e_{\boldsymbol{\gamma}}(\mathbf{z}, \boldsymbol{\Delta}) \equiv \sup_{\|f\|_{\boldsymbol{\gamma}} \leq 1} |I[f] - Q_n[f]|. \quad (6)$$

Here we use the notation $e_{\boldsymbol{\gamma}}(\mathbf{z}, \boldsymbol{\Delta})$ in place of $e_{\boldsymbol{\gamma}}(\mathbf{t}_0, \ldots, \mathbf{t}_{n-1})$ to refer to the worst case error of the point set generated by $\mathbf{z}$ (the generating vector) and $\boldsymbol{\Delta}$ (the random shift vector). The shift averaged worst case error, $e^{\mathrm{sh}}_{\boldsymbol{\gamma}}(\mathbf{z})$, is given by averaging the worst case error over uniformly distributed shifts in $[0, 1]^d$. The choice of weights, $\boldsymbol{\gamma}$, affects both the norm of the function and the worst case error. Choosing large weights leads to a smaller norm but larger worst case error and vice versa.

First we consider a Sobolev space spanned by functions f with square integrable (weak) derivatives $\frac{\partial^{|\mathbf{a}|} f(\mathbf{x})}{\partial \mathbf{x}_{\mathbf{a}}}$ and $\mathbf{a} \in \{0, 1\}^d$. The norm of f in the weighted Sobolev space can be written as

$$\|f\|^2_{\mathrm{Sobolev},\boldsymbol{\gamma}} = \sum_{\mathrm{u} \subseteq \{1,\ldots,d\}} \frac{1}{\gamma_{\mathrm{u}}} \int_{[0,1]^{|\mathrm{u}|}} \left(\int_{[0,1]^{d-|\mathrm{u}|}} \frac{\partial^{|\mathrm{u}|} f(\mathbf{x})}{\partial \mathbf{x}_{\mathrm{u}}} \mathrm{d}\mathbf{x}_{-\mathrm{u}} \right)^2 \mathrm{d}\mathbf{x}_{\mathrm{u}}. \quad (7)$$

Note that the norm depends only on the mixed *first* derivative because we never differentiate more than once with respect to a particular variable. It can be shown [39,46] that for functions belonging to such a space a R1SL rule exists for which the shift averaged worst case error is given by

$$[e^{\mathrm{sh}}_{\mathrm{Sobolev}}(\mathbf{z})]^2 \leq \left(\frac{1}{\psi(n)} \sum_{\emptyset \neq \mathrm{u} \subseteq \{1,\ldots,d\}} \gamma^{\lambda}_{\mathrm{u}} \left(\frac{2\zeta(2\lambda)}{(2\pi^2)^{\lambda}} \right)^{|\mathrm{u}|} \right)^{\frac{1}{\lambda}} \quad (8)$$

for all $\lambda \in (1/2, 1]$. Here ζ is the Riemann zeta function and ψ is the Euler totient function. This formula indicates that, for suitably chosen weights, R1SL rules can have a convergence rate close to $\mathcal{O}(n^{-1})$ independently of d for functions belonging to a Sobolev space.

The Korobov function space is a space of periodic functions which are α times differentiable in each variable. The parameter α is known as the smoothness parameter and characterizes the rate of decay of the Fourier coefficients of the integrand. The norm of f in the weighted Korobov space is given by

$$\|f\|^2_{\text{Korobov},\boldsymbol{\gamma}} = \sum_{\mathbf{h}\in\mathbb{Z}^d} \frac{\prod_{j\in \mathfrak{u}(\mathbf{h})} |h_j|^{2\alpha}}{\gamma_{\mathfrak{u}(\mathbf{h})}} |\hat{f}(\mathbf{h})|^2, \tag{9}$$

where $\mathfrak{u}(\mathbf{h}) := \{j \in \{1, \ldots, d\} : h_j \neq 0\}$ and $\hat{f}(\mathbf{h})$ are the Fourier coefficients of the integrand, given by

$$\hat{f}(\mathbf{h}) = \int_{[0,1]^d} f(\mathbf{x}) e^{-2\pi i \mathbf{h}\cdot\mathbf{x}} \mathrm{d}\mathbf{x}. \tag{10}$$

For functions belonging to a Korobov space with smoothness α the shift averaged worst case error is given by

$$[e^{\text{sh}}_{\text{Korobov}}(\mathbf{z})]^2 \leq \left(\frac{1}{\psi(n)} \sum_{\emptyset\neq\mathfrak{u}\subseteq\{1,\ldots,d\}} \gamma_{\mathfrak{u}}^{\lambda} \left(2\zeta(2\alpha\lambda)\right)^{|\mathfrak{u}|} \right)^{\frac{1}{\lambda}}, \tag{11}$$

for all $\lambda \in (1/(2\alpha), 1]$. The best convergence rate is obtained when $\lambda \to 1/(2\alpha)$, which yields a convergence close to $\mathcal{O}(n^{-\alpha})$ independently of d (for suitably chosen weights). Functions which are smooth but not periodic can be periodized by an integral transform as described in Section 2.3. This can improve the rate of convergence of quasi-Monte Carlo integration but may also increase the variance (or norm) of the function, especially in high dimensions.

2.2. Generating vectors

The convergence of the rank-1 lattice rule given in Eq. (2) depends on the choice of the generating vector $\mathbf{z}$ and in particular the worst case errors given in Eqs. (8) and (10) can only be achieved with specific choices of $\mathbf{z}$. An efficient algorithm to construct good generating vectors $\mathbf{z}$ is the component-by-component construction [47], where a generating vector in d dimensions is obtained from a $d-1$ dimensional one by selecting the additional component such that the worst-case error is minimal. This allows to construct the generating vectors with a cost of $\mathcal{O}(d\, n \log n)$.

We provide generating vectors for different fixed lattice sizes n, which have been obtained for a Korobov space with product weights $\gamma_{\mathfrak{u}} = \prod_{i\in\mathfrak{u}} \gamma_i$, where we set all weights equal, $\gamma_i = 1/d$. More details on the generating vectors provided with the QMC library are given in Appendix A.3.

2.3. Transformations

Lattice rules perform particularly well for continuous and smooth functions which are periodic with respect to each variable. Sector decomposed functions are typically continuous and smooth but not periodic. However, they can be periodized by a suitable change of variables $\mathbf{x} = \phi(\mathbf{u})$,

$$I[f] \equiv \int_{[0,1]^d} \mathrm{d}\mathbf{x}\, f(\mathbf{x}) = \int_{[0,1]^d} \mathrm{d}\mathbf{u}\, \omega_d(\mathbf{u}) f(\phi(\mathbf{u})) \tag{12}$$

where

$$\phi(\mathbf{u}) = (\phi(u_1), \ldots, \phi(u_d)),\ \omega_d(\mathbf{u}) = \prod_{j=1}^{d} \omega(u_j) \quad \text{and} \quad \omega(u) = \phi'(u). \tag{13}$$

In practice, the periodizing transform may be specified in terms of the weight function, ω, in which case the change of variables is given by

$$\phi(u) \equiv \int_0^u \mathrm{d}t\ \omega(t). \tag{14}$$

We have implemented the following periodizing transformations:

- Korobov transforms [48–50],
- Sidi transforms [51],
- Baker's transform [52].

The Korobov transform is defined by the polynomial weight function

$$\omega_{r_0,r_1}(u) = \frac{u^{r_0}(1-u)^{r_1}}{\int_0^1 \mathrm{d}t\ t^{r_0}(1-t)^{r_1}} = (r_0+r_1+1)\binom{r_0+r_1}{r_0} u^{r_0}(1-u)^{r_1}, \tag{15}$$

The weight parameters r_0, r_1 are usually chosen to be equal. The behaviour of the integrand near the endpoints should be taken into account when choosing the weight parameters r, as the variance of the integral can depend critically on their choice. Asymmetric Korobov transforms with $r_0 \neq r_1$ can be beneficial in cases where the integrand approaches a singularity near one of the endpoints, while the other endpoint does not exhibit any singular behaviour.

Sidi transforms [49,51] are trigonometric integral transforms with a weight proportional to $(\sin \pi u)^r$:

$$\omega_r(u) = \frac{(\sin \pi u)^r}{\int_0^1 \mathrm{d}t\ (\sin \pi t)^r} = \frac{\pi}{2^r} \frac{\Gamma(r+1)}{\Gamma((r+1)/2)^2} (\sin \pi u)^r. \tag{16}$$

The Sidi transforms may be used to periodize an integrand in a similar manner to the Korobov transforms. One potentially negative feature of the Sidi transforms is that several trigonometric functions need to be computed for each sample of the integrand. This can increase the cost (in terms of machine operations) considerably, especially for relatively simple integrands.

The baker's transformation [52] (also called "tent transformation"), given by

$$\phi(u) = 1 - \left|2u - 1\right| = \begin{cases} 2u & \text{if } u \leq \frac{1}{2}, \\ 2 - 2u & \text{if } u > \frac{1}{2}, \end{cases} \tag{17}$$

can be applied to achieve close to $\mathcal{O}(n^{-2})$ convergence for non-periodic integrands. The transform periodizes the integrand by mirroring rather than forcing it to a particular value on the integration boundary. Naively the fact that the transform is discontinuous might lead us to expect a poor asymptotic scaling (due to the fact that the transform is not smooth). However, an analysis based on considering the transform as a modification of the lattice rather than of the integrand allows the convergence of $\mathcal{O}(n^{-2})$ to be proven. In a moderate number of dimensions ($d \gtrsim 9$) the baker's transform typically does not increase the variance of the integrand as much as the Korobov and Sidi transforms. Therefore, although it has a slower convergence rate, the baker's transform can still prove useful.

A critically important point to consider when choosing a periodization strategy is the number of dimensions in which the integration will be performed. In particular, applying a periodizing transform can increase the variance of the integrand exponentially with its dimension d. Although it is possible to construct rank-1 lattice rules whose worst case error is independent of d (or depends at most polynomially on d), increasing the

variance of the integrand can spoil the convergence of the quasi-Monte Carlo integration [50]. For integrands in a relatively low number of dimensions ($d \lesssim 8$) the increase in variance caused by higher weight ($r \gtrsim 3$) periodizing transforms can be counteracted by the improved smoothness of the integrand which leads to an improved asymptotic scaling behaviour with the number of lattice points n.

2.4. Variance reduction

By applying a variable transformation $y = p(x)$ to a one-dimensional integral

$$I = \int_0^1 \mathrm{d}y\, f(y) = \int_0^1 \mathrm{d}x\, p'(x) f(p(x)), \tag{18}$$

the integration becomes trivial if $p'(x) \propto f(p(x))^{-1}$. While it is usually not possible to find a transformation fulfilling this condition exactly, it is possible to find approximations to it. This leads to an integrand with reduced variance, which can significantly improve the convergence of the integration when using numerical integration techniques. A well known method to apply variance reduction to multi-dimensional integrals is the VEGAS algorithm [53], where the above procedure is applied to each integration variable separately, with the remaining variables integrated out. In the algorithm of Ref. [53], for each integration variable x, the transformation $p(x)$ is constructed as a strictly increasing, piecewise linear function, such that $p'(x)$ resembles the shape of $|f(p(x))|^{-1}$. However, this procedure leads to discontinuities in $p'(x)$, which spoil the smoothness of the integrand and thus the scaling of the numerical integration when directly applying this algorithm in combination with QMC integration. Instead, we use the ansatz

$$p(x) = a_2 \cdot x \frac{a_0 - 1}{a_0 - x} + a_3 \cdot x \frac{a_1 - 1}{a_1 - x} + a_4 \cdot x + a_5 \cdot x^2 + \left(1 - \sum_{i=2}^{5} a_i\right) \cdot x^3 \tag{19}$$

to parametrize the variance reducing transformation. The parameters a_i are obtained via a fit to the inverse of the cumulative distribution function (CDF),

$$\mathrm{CDF}_f(x) = \int_0^x \mathrm{d}y\, |f(y)| \Big/ \int_0^1 \mathrm{d}y\, |f(y)|. \tag{20}$$

The ansatz in Eq. (19) is chosen such that $p(0) = 0$ and $p(1) = 1$. The parameters a_2 and a_3 are required to be positive, and $a_0 \in [1.001, \infty)$, $a_1 \in (-\infty, -0.001]$ such that no singularities are introduced within the domain of integration by the transforms. The parameters are optimized by sampling the integrand with a lattice of given size to numerically obtain an estimate of the CDF for each integration parameter and applying a non-linear least-squares fit using the routines implemented in the GNU Scientific Library [54].

We find that the ansatz in Eq. (19) works well for typical functions obtained by sector decomposition. While this ansatz in principle can be applied to other integrals as well, we expect that for other functions it can be beneficial to modify it to improve the fit of the CDF of the corresponding integrand.

3. Stand-alone usage of the integrator library

3.1. Installation

If you wish to use the integrator with your own code rather than within pySECDEC, then it is available as a c++11 single-header header-only library at https://github.com/mppmu/qmc. Download the header and include it in your project. Since the QMC is a header only c++ template library it does not need to be separately configured and built.

In order to build the header as part of your project you will need:

- A c++11 compatible c++ compiler.
- The GNU Scientific Library (GSL), version 2.5 or greater.
- (Optional GPU support) A CUDA compatible compiler (typically `nvcc`).
- (Optional GPU support) CUDA compatible hardware with Compute Capability 3.0 or greater.

Simply include it in your project, ensure that it can be found by your compiler (using compiler include path specifiers if necessary) and then build your project, linking against the GSL.

3.2. Minimal example

In this section we provide examples of the usage of the integrator as a stand-alone package. The usage within the c++ interface to pySECDEC is similar while the usage via the python interface to pySECDEC differs significantly. Both uses within pySECDEC are described in Section 4.2.

The code of a minimal program demonstrating the usage of the integrator is shown in Fig. 2. In this example, the 3-dimensional function $f(x_0, x_1, x_2) = x_0 x_1 x_2$ is integrated using the default settings of the QMC. Assuming the code is in a file named `minimal.cpp` and the QMC header can be found by the compiler, the program can be compiled without GPU support using the command:

```
c++ -std=c++11 minimal.cpp -o minimal.out -lgsl -lgslcblas
```

or with GPU support using the command:

```
nvcc -arch=<arch> -std=c++11 -x cu -Xptxas -O0 -Xptxas
--disable-optimizer-constants minimal.cpp -o minimal.out -lgsl -lgslcblas
```

where `<arch>` is the architecture of the target GPU or `compute_30` for just-in-time compilation (see the Nvidia `nvcc` manual for more details). The compile flag `-x cu` explicitly specifies the language of the input files as CUDA, rather than letting the compiler choose a default based on the file name suffix. The compile flag `-Xptxas -O0` disables optimization of the code by the PTX assembler, as of CUDA 9.2 we found rare cases where code optimization led to wrong results. The flag `-Xptxas --disable-optimizer-constants` disables the use of the optimizer constant bank which can be exhausted for large integrands, it is not strictly necessary to pass this flag for simple examples.

In Fig. 2, on lines 4–13, a functor `my_functor`, containing the function to be integrated is defined and instantiated. On line 19 the QMC integrator is instantiated with a Korobov transform of weight 3. The `MAXVAR` variable controls the maximum number of integration variables over which a particular instance of the QMC integrator can integrate, it should be set to a value equal to or larger than the maximum number of integration parameters present in any functor that will be passed to the instance of the QMC integrator. On line 20 the functor instance is passed to the `integrate` function of the integrator, this will trigger the numerical integration. The integrator returns a `result` struct containing the integral and its uncertainty, which are printed on lines 21–22. The CUDA function execution space specifiers `__host__` and `__device__` on line 7 are present only when compiling with GPU support. This is controlled by the presence on line 6 of the `__CUDACC__` macro which is automatically defined by the compiler during CUDA compilation.

```
#include <iostream>
#include "qmc.hpp"

struct my_functor_t {
    const unsigned long long int number_of_integration_variables = 3;
#ifdef __CUDACC__
    __host__ __device__
#endif
    double operator()(double* x) const
    {
        return x[0]*x[1]*x[2];
    }
} my_functor;

int main() {

    const unsigned int MAXVAR = 3;

    integrators::Qmc<double,double,MAXVAR,integrators::transforms::Korobov
        <3>::type> integrator;
    integrators::result<double> result = integrator.integrate(my_functor);
    std::cout << "integral = " << result.integral << std::endl;
    std::cout << "error = " << result.error << std::endl;

    return 0;
}
```

Fig. 2. A minimal example of the use of the QMC integrator.

```
integrators::Qmc<double,double,5,integrators::transforms::Korobov<3>::type
                ,integrators::fitfunctions::PolySingular::type // optional
               > integrator;
integrator.epsrel = 1e-5; // requested relative accuracy
integrator.epsabs = 1e-20; // requested absolute accuracy
integrator.maxeval = 10000000; // maximum number of function evaluations
integrators::result<double> result = integrator.integrate(my_integrand);
```

Fig. 3. Case 1 usage example of the QMC integrator.

3.3. Usage

We envisage two typical use case scenarios for the QMC library:

(1) The user knows relatively little about the integrand but wishes to know the result with a specific relative and/or absolute accuracy.
(2) The user has a reasonable idea how the QMC performs on their integrand and wishes to obtain a result as quickly as possible.

We discuss these use cases in turn. A description of all public fields and member functions is given in Appendix.

3.3.1. Case 1

In order to evaluate an integral to a specific relative and/or absolute accuracy without significant human input, the following QMC integrator member variables are relevant: `epsrel`, `epsabs`, `maxeval` along with the member function `integrate`.

Firstly, the QMC must be initialized with a suitable integral transform and the user must decide whether to use the variance reduction methods described in Section 2.4. If nothing is known about the periodicity and variance of the integrand, we would typically recommend using a Korobov weight 3 transform if the integrand lives in less than 9 dimensions (otherwise the baker transform may be more suitable) and no variance reduction. If the QMC does not produce even a rough estimate of the integral ($\sim$ 20% error) with a moderate lattice size then the variance reduction procedure may prove useful and the fit function should be specified as shown in Fig. 3.

The user can then set the `epsrel` and `epsabs` fields to the desired accuracy. In addition, the parameter `maxeval` ensures that the integration terminates in a reasonable time, even if the desired accuracy cannot be reached. The integration terminates once any of the three conditions is met. What constitutes a suitable value of `maxeval` depends on the complexity of the integrand (in terms of floating point operations), the hardware available for computing the integral and the time the user is willing to wait for a result.

Finally, the `integrate` function can be called on the input function. In Fig. 3 we display the above steps in code (for a 5-dimensional real integrand named `my_integrand`).

If a fit function has been provided, the QMC library will evaluate `evaluateminn` lattice points and use them as input to the fitting and variance reduction procedure as described in Section 2.4. The QMC library will then apply the selected periodizing transform to the fitted function. If no fit function has been provided the QMC will apply the periodizing transform to the input function and proceed to the next step directly.

In the next step, a total of `minm` randomly shifted copies of the smallest possible lattice greater than `minn` in size will be sampled and used to estimate the integration error. If the required error goal has not been reached the result will be discarded and a larger lattice will be selected and computed. This procedure will be repeated as necessary until the desired error goal is reached or `maxeval` function evaluations have been performed; at which

```
1 integrators::Qmc<double,double,5,integrators::transforms::Korobov<3>::type>
    integrator;
2 integrator.minn = 10000; // suitable lattice size
3 integrator.maxeval = 1; // do not evaluate larger lattices to satisfy default
    epsrel and epsabs
4 integrators::result<double> result = integrator.integrate(my_integrand);
```

Fig. 4. Case 2 usage example of the QMC integrator.

point the integration will terminate and the last result obtained will be returned.

If, during the iteration, the QMC requires a lattice larger than can be produced with the available generating vectors it will instead select the largest lattice and attempt to reduce the integration error by adding random shifts. In this case the QMC will achieve only Monte Carlo $\mathcal{O}(n^{-1/2})$ scaling. If an acceptable result cannot be achieved with Monte Carlo scaling then the user is advised to compute and supply additional (larger) generating vectors as described in Appendix A.3.

Note that, unlike some other integration algorithms, the results from all but the last iteration have no effect on the final result. It is therefore always more efficient to directly evaluate a lattice that gives an acceptable integration error rather than asking the library to try to find a suitable lattice size by iterating.

3.3.2. Case 2

If the user has a reasonable idea how the QMC performs on their integrand, for example by studying similar integrands or evaluating their integrand with a small lattice, then a result can most quickly be obtained by setting the parameters `minn` and calling the member function `integrate`. In order to ignore the default error goals `epsrel` and `epsabs`, the parameter `maxeval` should be set to 1. In Fig. 4 we display the above steps in code (for a 5-dimensional real integrand named `my_integrand`).

The QMC library will evaluate `minm` randomly shifted copies of the smallest possible lattice with at least `minn` points and return the result. If this result is not satisfactory then the user can increase `minn` and retry the integration. In order to estimate what lattice size is suitable it is sometimes useful to investigate the scaling behaviour of the integrand by evaluating several different lattices. Note that, as can be seen in Fig. 14, the scaling of the QMC is quite 'noisy' in the sense that very similarly sized lattices can produce estimates of the integral with errors that differ by an order of magnitude or more. This behaviour can hinder straightforward attempts to estimate the scaling behaviour of an integrand.

3.3.3. Usage on GPUs

In order to use the QMC library on CUDA enabled GPUs the user must ensure that their integrand functor can be evaluated on the chosen device. This usually entails taking the following steps:

- Ensuring that the `c++` language features used in the integrand function are supported by the relevant CUDA device.
- Designing the integrand function so that it does not need to access data that will be stored only in host memory.
- Marking the call operator of the integrand functor `__host__ __device__`, as shown in the examples above.

For the purpose of monitoring GPU usage and debugging we have found the following tools provided by Nvidia, and distributed with the CUDA toolkit, to be useful:

- `nvidia-smi`, a `top` like management and monitoring utility for Nvidia GPU devices.
- `cuda-memcheck`, a functional correctness checking suite.

In most cases the usage of GPUs within the QMC is straightforward, however, the attentive user may notice that the program behaves in a slightly different manner than when using only CPUs. Let us discuss some of the most prominent features of CUDA devices which can affect the usage of the QMC library.

The Nvidia kernel mode driver must be running and connected to the GPU device before any user interaction with that device can take place. If the kernel mode driver is not already running and connected to the target GPU the invocation of a program that interacts with the GPU will cause the driver to load and initialize the GPU. This will incur a start up cost of 1–3 s per GPU. For short running integration jobs this cost can be a significant fraction of the integration time. On Windows, the kernel mode driver is loaded at start up and kept loaded until shut down, however, by default the time-out detection and recovery (TDR) feature will cause driver reload and should be disabled (we refer to the latest Nvidia documentation). Similarly, under Linux, if an `X`-like process is run from start up to shut down it will usually initialize and keep alive the kernel mode driver. However, if no long-lived `X`-like client is kept running (for example in many HPC environments) the kernel mode driver will initialize and de-initialize the target GPU each time a GPU application starts and stops. As of CUDA 9.2 the Nvidia recommended way to circumvent the delay due to starting and stopping the kernel mode driver is to run the Nvidia Persistence Daemon (we refer to the latest Nvidia documentation).

When compiling with `nvcc` we strongly recommend to look up and enter the *real* architecture of the graphics card in use, e.g. `-arch=sm_70`. If a *virtual* architecture is specified, the device code is just-in-time compiled for the *real* architecture on a single core, which may become the dominant fraction of the runtime. For initial tests, the *virtual* architecture `compute_30`, which is the oldest supported in CUDA version 9.2, should be compatible with most GPUs that are currently in use. For more information we refer to the `nvcc` manual.

3.4. Design and implementation

In order to numerically integrate a single function, the QMC integrator library can concurrently utilize multiple multi-threaded CPUs as well as multiple CUDA hardware accelerators, provided they all belong to a single system. To achieve reasonable performance on heterogeneous systems a receiver-initiated central work queue load balancing algorithm is utilized.

The load balancing algorithm consists of the following steps:

- A central work queue is initialized by the main thread.
- The main thread then spawns `cputhreads` worker threads if -1 is listed in `devices` and additionally one worker thread per GPU is listed in `devices`.
- Each worker requests work from the work queue and when the work is completed continues to request work until the queue is cleared, at which point the workers terminate.

The disadvantage of this algorithm is that the central work queue must be atomically locked to ensure work is not repeated. This can impact performance significantly when a quick-to-evaluate integrand is computed using a large number of cores and/or

CUDA devices. The advantage of this design is that even if the performance of the workers differs vastly (for example a worker computing on a single CPU core compared to a worker distributing work to a powerful GPU) the workload is reasonably balanced provided that the work packages are not so large as to leave a poorly performing worker with so much work that it finishes significantly later than all others.

In order to utilize massively parallel CUDA hardware, threads assigned to provide work to an accelerator will request significantly more work from the queue per access than workers assigned to a single CPU core. The amount of work (number of "work packages") requested at once by a worker assigned to a CUDA device is controlled by the product of `cudablocks` and `cudathreadsperblock`, while workers assigned to a CPU core always request only a single "work package".

At the time of release the default values of parameters affecting the load balancing are usually a reasonable choice for most feasible integrands and existing hardware. Naturally, as the state of the art advances and computer hardware evolves we may alter these default values in future QMC releases.

4. Usage of the integrator library within pySecDec

Here we describe briefly the installation and usage of pySecDec, focusing on the usage of the QMC integrator. For more details we refer to the manual https://secdec.readthedocs.io and to the examples distributed with pySecDec.

4.1. Installation

Before installing pySecDec, make sure that recent versions of numpy (http://www.numpy.org/) and sympy (http://www.sympy.org/) are installed. The pySecDec program (which includes the QMC integrator library) can be downloaded from https://github.com/mppmu/secdec/releases. To install pySecDec, perform the following steps

```
tar -xf pySecDec-<version>.tar.gz
cd pySecDec-<version>
make
<copy the highlighted output lines into your .bashrc>
```

The `make` command will automatically build further dependencies in addition to pySecDec itself. Further notes on the installation procedure are summarized in the online documentation https://secdec.readthedocs.io. To get started, we recommend to read the section "getting started" in the online documentation.

4.2. Usage

Depending on the availability, it is possible to use the program with CPUs, GPUs or a combination of both.

4.2.1. Using CPUs only

The basic steps can be summarized as follows:

(1) Write or edit a `python` script to define the integral, the replacement rules for the kinematic invariants, the requested order in the regulator and some other options, see e.g. the example `examples/easy/generate_easy.py`.
(2) Run the script `generate_easy.py` using `python`. This will generate a subdirectory according to the `name` specified in the script.
(3) Type `make -C <name>`, where `<name>` is your chosen name. This will create the `c++` libraries.
(4) Write or edit a `python` script to perform the numerical integration using the `python` interface, see e.g. `examples/easy/integrate_easy.py`. Make sure that the QMC integrator is chosen in that file.

4.2.2. Using GPUs and CPUs

When using GPUs, steps (1), (2) and (4) of the previous Section 4.2.1 are the same. The only difference is in the compilation of the sector files

(3) Type `CXX=nvcc SECDEC_WITH_CUDA=<arch> make -C <name>`, where `<name>` is your chosen name and `<arch>` is the argument forwarded to `nvcc` as `-arch=<arch>`. This will create the `c++` libraries.

The compute capability `<arch>` is specific to each graphics card. The parameter `<arch>` can either be a suitable *virtual* architecture or a *real* architecture. We strongly recommend to look up and enter the *real* architecture of the graphics card in use, e.g. `sm_70`. If a *virtual* architecture is specified, the device code is just-in-time compiled for the *real* architecture on a single core, which may become the dominant fraction of the runtime. For first tests however, the *virtual* architecture `compute_30`, which is the oldest supported in CUDA version 9.2, should be compatible with most GPUs that are currently in use. For more information refer to the `nvcc` manual.

4.3. Examples

All the examples described below can be found in the folder `examples` of the pySecDec distribution. A comparison of the timings for the examples can be found in Table 3. The settings for the examples are default settings unless specified otherwise. The setting `maxeval=1` ensures the evaluation of the integrand with a fixed number of sampling points as described in Section 3.3.2.

4.3.1. Basic usage

The basic usage of pySecDec is illustrated in the example `easy`. The slightly modified `easy_cuda` example shows how to compile and run the easy example on all available GPUs using either the `python` or the `c++` interface.

The generate file `generate_easy.py` shown in Fig. 5 is identical for both examples, `easy` and `easy_cuda`. The integrate file `integrate_easy.py` differs by the optional lines that select the QMC integrator. Choosing the QMC integrator as shown in Fig. 5 will make pySecDec use all CPU cores as well as all available GPUs.

In order to use GPUs, the code should be compiled with Nvidia's `nvcc` compiler. It is also possible to use non-CUDA compilers, though this will disable GPU support.

The commands to run the examples are

(a) using CPUs and GPUs:

```
python generate_easy.py
CXX=nvcc SECDEC_WITH_CUDA=compute_30 make -C easy
python integrate_easy.py
```

or (b) using CPUs only:

```
python generate_easy.py
make -C easy
python integrate_easy.py
```

How to use the QMC integrator with GPU support via the `c++` interface is shown in the example `easy_cuda`. Note that above we have set the compute capability to `compute_30`. Please read the remarks about the compute capability in the previous section before running more complicated examples on the GPU.

4.3.2. 3-mass banana graph

The example `banana_3mass` calculates a three-loop two-point integral with three different internal masses, see Fig. 6. If the three masses are different, the analytic result cannot be expressed anymore by products of complete elliptic integrals [4], and therefore is an example of a "hyperelliptic" integral. The purpose of this example is to show that hyperelliptic integrals can

```python
from pySecDec import make_package

make_package(

name = 'easy',
integration_variables = ['x','y'],
regulators = ['eps'],

requested_orders = [0],
polynomials_to_decompose = ['(x+y)^(-2+eps)'],

)
```

(a) `generate_easy.py`

```python
from pySecDec.integral_interface import IntegralLibrary
from math import log

# load c++ library
easy = IntegralLibrary('easy/easy_pylink.so')

# choose Qmc integrator
# automatically uses all avaliable GPUs
easy.use_Qmc(transform='korobov3')

# integrate
_, _, result = easy()

# print result
print('Numerical Result:' + result)
print('Analytic Result:' + ' + (%.15g)*eps^-1 + (%.15g) + O(eps)' % (1.0,1.0-log(2.0)))
```

(b) `integrate_easy.py`

Fig. 5. pySecDec input for a simple integral.

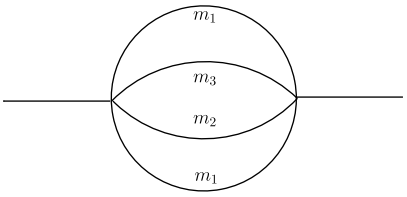

Fig. 6. A 3-loop 2-point function with 3 different masses.

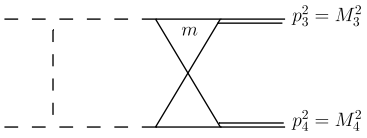

Fig. 7. A non-planar 2-loop 4-point function with a massive loop and two massive legs with different masses.

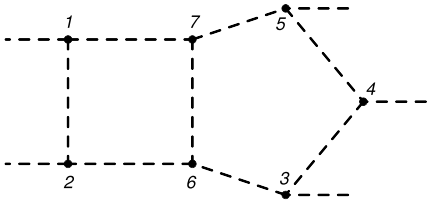

Fig. 8. A 2-loop pentabox integral in $d = 6 - 2\epsilon$.

be evaluated as fast as other integrals which are more accessible analytically.

The result for the non-Euclidean point $s = 20.0$, $m_1^2 = 1.0$, $m_2^2 = 1.3$, $m_3^2 = 0.7$, computed with the QMC and the settings `minn=1000000`, `maxeval=1`, `transform='korobov2'` reads

$$\begin{aligned} I = &(1.97000000000000264 \pm 9.85 \cdot 10^{-15} \\ &+ \mathrm{i}(1.84 \cdot 10^{-15} \pm 1.16 \cdot 10^{-15})) \cdot \epsilon^{-3} \\ &+ (-5.9281676367925620 \pm 6.52 \cdot 10^{-14} \\ &+ \mathrm{i}(1.07 \cdot 10^{-13} \pm 2.63 \cdot 10^{-14})) \cdot \epsilon^{-2} \\ &+ (9.86757086818429 \pm 1.64 \cdot 10^{-12} \\ &- \mathrm{i}(2.54 \cdot 10^{-11} \pm 9.83 \cdot 10^{-12})) \cdot \epsilon^{-1} \\ &- 89.066074732329 \pm 8.25 \cdot 10^{-10} \\ &+ \mathrm{i}(8.10892634289 \pm 2.37 \cdot 10^{-9}) \\ &+ \mathcal{O}(\epsilon) . \end{aligned} \tag{21}$$

The imaginary parts of the pole coefficients are numerically zero, the accuracy of this zero being limited by the fact that we are operating close to machine precision.

4.3.3. Non-planar 4-point function with massive propagators and massive legs of different mass

The example `HZ2L_nonplanar` calculates a non-planar four-point two-loop integral in the physical region, where one loop is fully massive, and two of the external legs are massive/off-shell with two different masses, see Fig. 7. The commands to run this example are analogous to the ones given above.

The result for the point $s = 200$, $t = -23$, $m^2 = 9$, $M_3^2 = 1.56$, $M_4^2 = 0.81$, obtained with the QMC integrator using the settings `minn=10**8`, `maxeval=1`, `transform='korobov3'` reads

$$\begin{aligned} &(3.4401552304457233 \cdot 10^{-6} \pm 1.73 \cdot 10^{-20} \\ &- \mathrm{i}(8.9 \cdot 10^{-23} \pm 2.26 \cdot 10^{-21})) \cdot \epsilon^{-2} \\ &+ (-0.00003316795824 \pm 1.16 \cdot 10^{-12} \\ &- \mathrm{i}(8.53692099 \cdot 10^{-6} \pm 1.01 \cdot 10^{-12})) \cdot \epsilon^{-1} \\ &+ 0.000159345747 \pm 4.12 \cdot 10^{-10} \\ &+ \mathrm{i}(0.000021017686 \pm 3.89 \cdot 10^{-10}) \\ &+ \mathcal{O}(\epsilon) . \end{aligned} \tag{22}$$

4.3.4. Pentabox

The example `pentabox_fin` calculates a fully massless two-loop five-point function in the physical region with $d = 6 - 2\epsilon$, see Fig. 8.

The pentabox is a master integral occurring in the calculation of $2 \to 3$ scattering at two loops. In $4 - 2\epsilon$ dimensions, sector decomposition produces poles of order ϵ^{-5} at intermediate

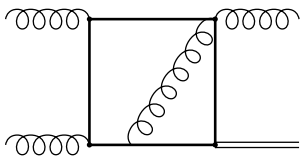

Fig. 9. A 2-loop 2-point integral appearing at NLO in Higgs plus jet production.

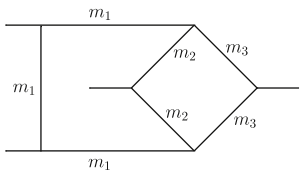

Fig. 10. A 2-loop integral leading to hyperelliptic functions [56].

stages. The $6-2\epsilon$ dimensional version we investigate here is finite and therefore a more suitable master integral for numerical evaluation. This is an example where the use of a fit function considerably improves the convergence.

The result for the non-Euclidean point $s_{12}=5$, $s_{23}=-4$, $s_{34}=2$, $s_{45}=-6$ and $s_{51}=3$, obtained with the QMC integrator using the settings `minn=10**8`, `maxeval=1`, `transform='korobov3'`, `fitfunction='polysingular'` reads

$$P = -0.0198236478 \pm 2.02 \cdot 10^{-8} - \mathrm{i}\,(\,0.0341514614 \pm 1.59 \cdot 10^{-8}\,) + \mathcal{O}(\epsilon)\,. \tag{23}$$

4.3.5. Elliptic 2-loop integral

The example `elliptic2L_physical` calculates a planar two-loop four-point function with one off-shell leg and a massive loop in the physical region, see Fig. 9. This diagram enters the NLO corrections to Higgs+ jet production and contains elliptic structures. The analytical result in the Euclidean region is given in Ref. [55]. While a numerical result for this integral already has been given in Ref. [35], the purpose of this example is to demonstrate that the number of correct digits which can be obtained using the QMC integrator cannot be reached in a reasonable amount of time using Monte Carlo integration.

The result for the non-Euclidean point $s=90$, $t=-2.5$, $p_4^2=1.6$, $m^2=1$ using Vegas reads

$$f_{66}^{A}\cdot\left(\frac{-s}{m^2}\right) = -0.044289 \pm 2.5 \cdot 10^{-5} + \mathrm{i}\,(\,0.016068 \pm 2.7 \cdot 10^{-5}\,) + \mathcal{O}(\epsilon)\,. \tag{24}$$

Using the QMC integrator with `minn=2147483647`, `maxeval=1`, `transform='korobov1'`, `fitfunction='polysingular'`:

$$f_{66}^{A}\cdot\left(\frac{-s}{m^2}\right) = -\,0.04429245890863 \pm 1.82 \cdot 10^{-13} + \mathrm{i}\,(\,0.01607147782349 \pm 1.69 \cdot 10^{-13}\,) + \mathcal{O}(\epsilon)\,.$$

4.3.6. Hyperelliptic 2-loop integral

The example `hyperelliptic` calculates a non-planar two-loop four-point function with three different masses and all propagators massive in the physical region, see Fig. 10. This integral is special since it is extremely hard to compute analytically, but is easily accessible numerically.

The result for the non-Euclidean point $s=10$, $t=-0.75$, $m_1^2=1$, $m_2^2=1.3$, $m_3^2=0.7$, computed with the QMC integrator using `minn=10**8`, `maxeval=1`, `transform='korobov3'`, `fitfunction='polysingular'` reads

$$I = -0.009449626 \pm 1.54 \cdot 10^{-7} + \mathrm{i}\,(\,0.019368308 \pm 1.60 \cdot 10^{-7}\,) + \mathcal{O}(\epsilon)\,. \tag{25}$$

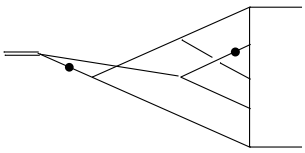

Fig. 11. A 4-loop massless form factor integral [57].

4.3.7. 4-loop form factor example

The example `formfactor4L` calculates a four-loop three-point integral in $d=6-2\epsilon$, see Fig. 11. Its analytic result is given in Eq. (7.1) of Ref. [57]. This example demonstrates the power of pySecDec to perform an efficient sector decomposition, even for integrals with many loops and internal propagators. Furthermore, it is a prime example to show how the QMC algorithm works for a larger number of integration dimensions (in this case 11 dimensions).

Since the integral has only one scale, the latter can be factorized. For better comparison with Ref. [57], we set the scale to -1 and the prefactor to $(\Gamma(d/2-1))^4$. Note that a factor $(\mathrm{i}\pi^{d/2})^{-L}$, where L is the number of loops, is part of the integral measure used in pySecDec, such that the prefactor corresponds to Eq. (2.4) of Ref. [57].

The result using the QMC integrator with `minn=35*10**5`, `minm=64`, `maxeval=1`, `cudablocks=128`, `cudathreadsperblock=64`, `maxnperpackage=8`, `maxmperpackage=8`, `verbosity=3`, `transform='baker'`, `fitfunction='polysingular'`

$$F^{\text{num.}} = +\,(3.1807379885 \pm 9.19 \cdot 10^{-8}) + (46.10430477 \pm 1.34 \cdot 10^{-6})\cdot\epsilon + \mathcal{O}(\epsilon^2)\,, \tag{26}$$

which can be compared to the analytical result of Ref. [57]

$$F^{\text{analyt.}} = 3.1807380843134699650 + 46.104303262308462367\epsilon + \mathcal{O}(\epsilon^2)\,. \tag{27}$$

To achieve an approximate $1/n$ scaling behaviour, the Baker transform had to be applied to the integrand. For this 11-dimensional parameter integral, the Baker transform is superior to the Korobov transform as it does not increase the variance of the integrand. For details we refer to Ref. [58].

4.3.8. 2-loop nbox

The Nbox example[1] consists of three integrals, `Nbox2L_split_a`, `Nbox2L_split_b`, and `Nbox2L_split_c`, all of which have one massive internal line that matches the mass of one external leg. The integral `Nbox2L_split_a` is shown in Fig. 12(a), `Nbox2L_split_b` is represented in Fig. 12(b) and `Nbox2L_split_c` by Fig. 12(b) with the dot removed. These integrals are of interest since they have no Euclidean region and thus the sector decomposition algorithms implemented in pySecDec are not guaranteed to succeed. In practice, the integral `Nbox2L_split_a` and `Nbox2L_split_b` can be computed using the `split=True` option of pySecDec. The integral `Nbox2L_split_c` is quasi-finite and therefore does not need `split=True`.

[1] Inspired by a private communication with H. Frellesvig and K. Kudashkin.

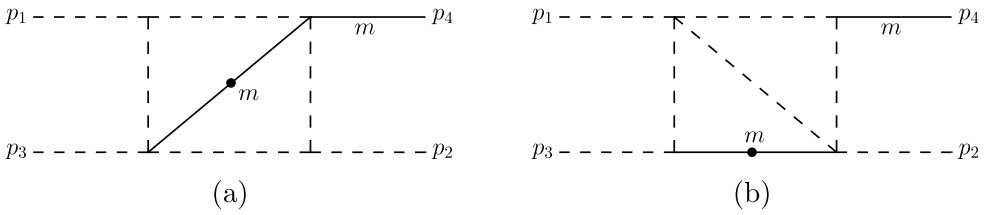

Fig. 12. 2-loop four-point integrals with one massive propagator and one massive leg.

The result for the point $s = (p_1 + p_2)^2 = -1$, $t = (p_1 + p_3)^2 = -0.8$ and $m^2 = 0.1$ is

$$\begin{aligned} H_a = &+ (1.54320987654321673 \cdot 10^{-1} \pm 7.04 \cdot 10^{-16} \\ &+ \mathrm{i}(3.95 \cdot 10^{-18} \pm 1.41 \cdot 10^{-16})) \cdot \epsilon^{-3} \\ &+ (-2.6079701365346328 \pm 1.18 \cdot 10^{-14} \\ &+ \mathrm{i}(1.93925472443813307 \pm 7.96 \cdot 10^{-15})) \cdot \epsilon^{-2} \\ &+ (-3.73711324653151 \pm 1.82 \cdot 10^{-12} \\ &- \mathrm{i}(9.93265048220209 \pm 1.13 \cdot 10^{-12})) \cdot \epsilon^{-1} \\ &+ 36.882907731123 \pm 2.42 \cdot 10^{-10} \\ &- \mathrm{i}(27.77041218391 \pm 8.59 \cdot 10^{-10}) \\ &+ \mathcal{O}(\epsilon)\,, \end{aligned} \tag{28}$$

$$\begin{aligned} H_b = &+ (-8.1789971643514132 \pm 4.96 \cdot 10^{-14} \\ &- \mathrm{i}(1.71 \cdot 10^{-15} \pm 2.80 \cdot 10^{-14})) \cdot \epsilon^{-2} \\ &+ (-3.0495945501 \pm 7.22 \cdot 10^{-8} \\ &- \mathrm{i}(51.3901546473 \pm 6.58 \cdot 10^{-8})) \cdot \epsilon^{-1} \\ &+ 160.02687326 \pm 2.83 \cdot 10^{-6} \\ &- \mathrm{i}(134.42897220 \pm 2.82 \cdot 10^{-6}) \\ &+ \mathcal{O}(\epsilon)\,, \end{aligned} \tag{29}$$

$$\begin{aligned} H_c = &+ 2.4083471021928 \pm 4.33 \cdot 10^{-11} \\ &- \mathrm{i}(25.8748336621213 \pm 4.59 \cdot 10^{-11}) \\ &+ \mathcal{O}(\epsilon)\,. \end{aligned} \tag{30}$$

To produce these results we have used the settings:

- `minn=10**7, maxeval=1, transform='korobov4', fitfunction='polysingular'` for H_a,
- `minn=10**9, maxeval=1, transform='korobov6', fitfunction='polysingular'` for H_b and
- `minn=15173222401, maxeval=1, transform='korobov6', generatingvectors='cbcpt_cfftw1_6'` for H_c.

4.3.9. 6-loop bubble

The `bubble6L` example consists of the 6-loop 2-point integral shown in Fig. 13. The pole coefficients are given analytically in Eq. (A3) of Ref. [59] (at $p^2 = -p_E^2 = -1$, where p_E is the external momentum in Euclidean space). The pySecDec symmetry finder reduces the number of sectors from more than 14 000 to 8774. We also note that the decomposition method `'geometric'` needs to be used, as the method `'iterative'` leads to an infinite recursion. The analytic result is given by

$$\begin{aligned} B_{6L}^{\text{analyt.}} = &\frac{1}{\epsilon^2}\frac{147}{16}\zeta_7 - \frac{1}{\epsilon}\left(\frac{147}{16}\zeta_7 + \frac{27}{2}\zeta_3\zeta_5 + \frac{27}{10}\zeta_{3,5}\right. \\ &\left. - \frac{2063}{504000}\pi^8\right) + \mathcal{O}(\epsilon^0) \end{aligned}$$

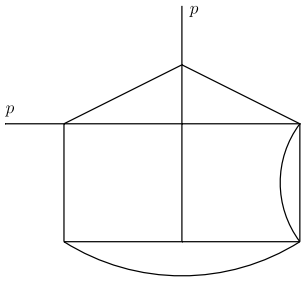

Fig. 13. A 6-loop two-point integral from Ref. [59].

$$= \frac{9.264208985946416}{\epsilon^2} + \frac{91.73175282208716}{\epsilon} + \mathcal{O}(\epsilon^0)\,. \tag{31}$$

The pySecDec result at $p^2 = -1$ obtained with the QMC integrator using `minn=10**7, maxeval=1, transform='baker', fitfunction='polysingular'` reads

$$\begin{aligned} B_{6L}^{\text{num.}} = &+ (9.2642089624 \pm 1.58 \cdot 10^{-8}) \cdot \epsilon^{-2} \\ &+ (91.73175426 \pm 2.15 \cdot 10^{-6}) \cdot \epsilon^{-1} \\ &+ (1118.607204 \pm 1.31 \cdot 10^{-4}) + \mathcal{O}(\epsilon)\,. \end{aligned} \tag{32}$$

5. Profiling

5.1. Scaling behaviour

Fig. 14 shows how the integration error of the QMC algorithm scales with the lattice size n for two different integrals. The plot on the left-hand side shows results for the $\mathcal{O}(\varepsilon^4)$ contribution of a 3-loop massless form-factor integral, which can be found in `examples/triangle3L` of the pySecDec distribution. Using the Korobov transformation with weight $\alpha = 1$ for the periodization (see Section 2.3), we obtain per-mille-level precision for $n = 1021$ and the integration error scales approximately as $\mathcal{O}(n^{-1})$, leading to a relative precision of 10^{-9} for $n \approx 10^9$. With a weight parameter $\alpha = 3$, we obtain slightly larger errors for small n, but due to a scaling with approximately $\mathcal{O}(n^{-2})$, a relative precision of 10^{-14} can be reached with $n \approx 10^9$. We note that the expected $\mathcal{O}(n^{-3})$ asymptotic scaling is not observed for lattices with $n \approx 10^8$ and that, due to the use of double precision arithmetic, the integration error does not decrease when choosing even larger lattice sizes. The plot also shows that increasing the lattice size does not always lead to a corresponding improvement of the integration error. Instead, for individual lattices, the integration error can be significantly larger than that obtained from a lattice of similar size. We observe this effect for nearly all integrals, but which lattices lead to relatively large uncertainties depends on the integrand.

The right-hand plot of Fig. 14 shows the results of an integral contributing to the NLO QCD corrections in Higgs + jet production [44] and has been selected as an example showing only slow

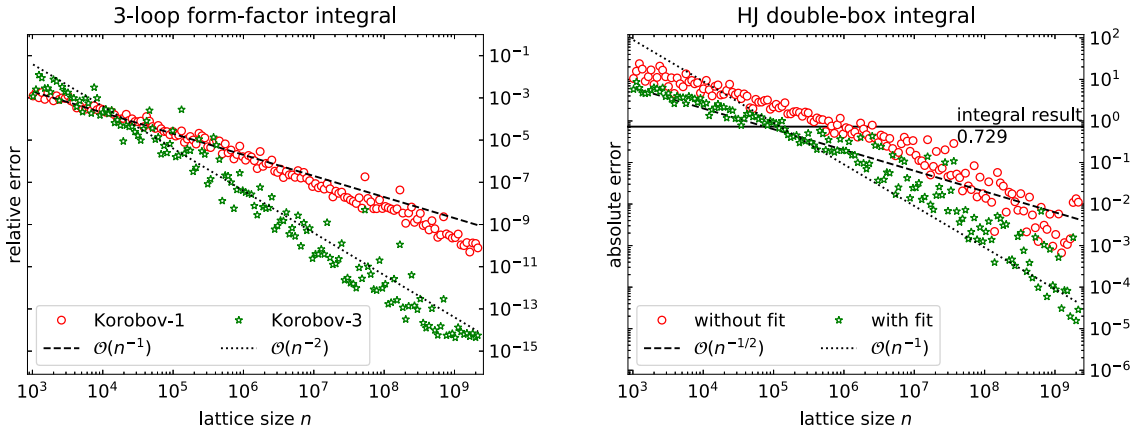

Fig. 14. Scaling of the integration error with the number of lattice points n for two different integrals. The left plot shows the relative error of the $\mathcal{O}(\varepsilon^4)$ contribution of a 3-loop form-factor integral using Korobov transformations of different weight. The plot on the right-hand side shows the scaling behaviour of a single sector of an integral, which appeared in an early stage of the calculation in Ref. [44], also demonstrating the effect of importance sampling.

convergence of the integration. The integrand is a single sector of a loop integral evaluated at a phase-space point with large invariant mass $m_{Hj} = 8.8\, m_t$ of the Higgs–jet system, using a Korobov transform with $\alpha = 1$ for the periodization. The code can be found in `examples/103_hj_double_box.cpp` of the QMC library. For lattice sizes $n \lesssim 10^6$, we observe that the integration error only scales with $n^{-1/2}$ and is larger than the true result of the integral. For larger lattice sizes, however, we find the expected $\mathcal{O}(n^{-1})$ scaling of the integration, allowing us to obtain the result with a precision better than 0.1%. Combining the QMC with importance sampling, the integration error for small lattice sizes is reduced by about a factor of 3 and improvements by more than a factor of 10 can be seen for large n. This example shows that sampling the integrand with a lattice of sufficient size is required to obtain the desired scaling of the QMC integration. We want to point out that for loop integrals it is possible to change the basis of required integrals using integration-by-part identities [60,61]. In many cases, this allows one to find a basis of integrals with an improved convergence of the numerical integration. For the results presented in Ref. [44], the integral discussed above was not used and, instead, it was possible to find an integral basis, where evaluating the corresponding integrals with $n \approx 10^6$ was sufficient to obtain accurate results.

5.2. Timings for test functions

For comparison of the performance of numerical integrators, Genz [62] has introduced a test suite, consisting of six integrand functions with different features:

1. Oscillatory $f_1(\mathbf{x}) = \cos(\sum_{i=1}^n c_i x_i + 2\pi w_1)$
2. Product peak $f_2(\mathbf{x}) = \prod_{i=1}^n (c_i^{-2} + (x_i - w_i)^2)^{-1}$
3. Corner peak $f_3(\mathbf{x}) = (1 + \sum_{i=1}^n c_i x_i)^{-(n+1)}$
4. Gaussian $f_4(\mathbf{x}) = \exp(-\sum_{i=1}^n c_i^2 (x_i - w_i)^2)$
5. C^0 Function $f_5(\mathbf{x}) = \exp(-\sum_{i=1}^n c_i |x_i - w_i|)$
6. Discontinuous $f_6(\mathbf{x}) = \begin{cases} 0 & \text{if } x_1 > w_1 \text{ or } x_2 > w_2, \\ \exp(\sum_{i=1}^n c_i x_i) & \text{otherwise .} \end{cases}$

The test functions help to identify how well an integrator handles oscillatory functions, multiple periodic peaks, one peak anywhere in the integration region, one peak at the end of the integration region, C^∞ functions, a continuous function whose derivatives are not continuous and finally a discontinuous function.

The integration region for all test integrands is the unit hypercube. The parameters w_i can be chosen randomly and should not affect the rate of convergence as long as $0 \leq w_i \leq 1$. On the contrary, the positive parameters $c_i > 0$ should affect the convergence behaviour, raising the complexity of the integral when $\|\mathbf{c}\|_1$ is increased.

We integrate each of the above functions with the parameters:

Number of dimensions:	$s = 5, 8, 10$
Requested relative accuracy:	$\epsilon_{\text{rel}} = 10^{-8}$
Maximum number of samples:	$n_{\max} = 7 \times 10^8$
Time limit:	$\lesssim 360$ s

All tests are performed on a machine with 2 x Intel Xeon Gold 6140 CPU @ 2.30 GHz CPUs (36 cores, 72 threads) and 4 x Nvidia Tesla V100 GPUs. In practice, the time limit only restricts the maximum number of samples used by the Suave integrator of Cuba to $\lesssim 8 \times 10^7$ samples. Without the time limit the Suave integrator would take up to 3000 s for some examples.

The integrand difficulties are set in accordance with Ref. [37] to:

Integrand family j	1	2	3	4	5	6
$\|\mathbf{c}\|_1$	6.0	18.0	2.2	15.2	16.1	16.4

For profiling we integrate each function 10 times in each dimension, setting the values of w_i and c_i randomly for each integration, and average the number of correct digits obtained and time taken. For all examples the QMC is instantiated with a weight 3 Korobov transform and the Cuba settings are set according to the test suite demo distributed with the latest version of Cuba (as of writing Cuba 4.2).

In Table 1 we show for each integrator the average number of digits obtained (calculated by comparing to the analytic result) and the time taken. Note that the Cuba integrators and the QMC (CPU) instance do not make use of the GPUs, whilst the integrator denoted QMC makes use of all CPUs and GPUs. Some of Cuba's algorithms sample the integrand serially for at least part of the numerical integration, which can greatly increase the time required to reach $n_{\max}$ evaluations. On the contrary, the QMC always samples in parallel and can usually make good use of all cores and devices. Furthermore, with the settings suggested by the test suite demo (distributed with Cuba), we find that our test bed machine is not well loaded by the Cuba integrators. The load produced by the Vegas algorithm of Cuba can be increased by up to a factor of 10 by increasing the `nstart` and/or `nincrease` settings and altering the `nbatch` setting.

The timings for the Divonne integrator of Cuba are omitted from Table 1. The Divonne algorithm, as implemented in Cuba,

Table 1
Number of correct digits computed (time in seconds) for the evaluation of the test integrands using the QMC and the integrators implemented in Cuba (Vegas, Suave and Cuhre).

Family	QMC (CPU)	QMC	Vegas	Suave	Cuhre
1 ($d = 5$)	9 (0.74)	9 (0.71)	6 (290)	5 (260)	9 (2.1)
1 ($d = 8$)	8 (29)	8 (1)	6 (300)	4 (270)	9 (2.1)
1 ($d = 10$)	6 (68)	6 (0.81)	6 (340)	4 (290)	9 (15)
2 ($d = 5$)	10 (3.6)	10 (0.6)	8 (320)	5 (280)	9 (26)
2 ($d = 8$)	6 (63)	6 (0.79)	7 (330)	5 (290)	8 (350)
2 ($d = 10$)	5 (53)	5 (0.84)	7 (340)	5 (290)	8 (180)
3 ($d = 5$)	11 (3.4)	11 (0.49)	7 (330)	5 (280)	8 (15)
3 ($d = 8$)	7 (56)	7 (0.75)	6 (340)	5 (290)	9 (21)
3 ($d = 10$)	6 (75)	6 (0.78)	6 (350)	4 (300)	8 (92)
4 ($d = 5$)	10 (3.8)	10 (0.63)	6 (330)	6 (280)	9 (110)
4 ($d = 8$)	6 (53)	6 (0.75)	6 (330)	5 (290)	9 (82)
4 ($d = 10$)	5 (68)	5 (0.78)	6 (340)	5 (290)	9 (140)
5 ($d = 5$)	8 (35)	8 (0.77)	8 (320)	5 (280)	6 (3000)
5 ($d = 8$)	5 (57)	5 (0.74)	7 (340)	5 (290)	4 (530)
5 ($d = 10$)	4 (71)	4 (0.79)	7 (340)	5 (290)	3 (190)
6 ($d = 5$)	5 (35)	5 (0.78)	4 (320)	3 (280)	4 (16)
6 ($d = 8$)	4 (63)	4 (0.72)	4 (340)	2 (290)	5 (73)
6 ($d = 10$)	3 (66)	3 (0.76)	5 (340)	2 (290)	6 (60)

Table 2
Number of correct digits computed for the evaluation of the test integrands in $d = 10$ using the QMC with the Baker transform.

Family ($d = 10$)	1	2	3	4	5	6
Digits	8	8	6	8	7	5

consists of 3 phases: (1) partitioning of the integration region, (2) sampling of the subregions and (3) refinement and resampling of the subregions. With the setting $\epsilon_{\text{rel}} = 10^{-8}$ and a time limit of $\lesssim$ 360 s the Divonne integrator usually does not complete the first phase and so does not enter the second phase. Without the second phase the Divonne integrator often underestimates the integration error and for the tests described in this section it typically returns results with only 2 correct digits. The Divonne integrator performs much more reliably with the setting $\epsilon_{\text{rel}} = 10^{-5}$, returning results accurate to 4–5 digits in around 20 s in all cases.

We reiterate the warning given in Ref. [37] that the comparison chart should be interpreted with care. In particular, we emphasize that the test integrands appearing in the test suite, by virtue of their simplicity, bear few similarities to integrands for which numerical integration is typically applied. For this suite of functions the Cuhre routine as implemented in Cuba performs best, this is due to the particular functions chosen for the test suite and is usually not the case when applying the integrators to sector decomposed functions. The QMC performs reasonably on the example functions, often beating the number of digits obtained by any of the other Cuba integrators and taking less time. The QMC performs worse, as expected, when integrating functions which are not smooth, in particular, the C^0 and discontinuous functions. When utilizing GPUs the QMC typically takes around 1 s to compute the samples (compared to 30–80 s for the CPU) regardless of the actual number of function evaluations. This indicates that the number of samples to be computed in these examples is too small to fully saturate the GPUs.

Relatively few correct digits are obtained by the QMC when integrating the examples with $d = 10$. One reason for this is the use of the weight 3 Korobov transform, which increases the variance of the integrand as described at the end of Section 2.3. In Table 2 we show the number of correct digits obtained using the QMC with the Baker transform (rather than the Korobov transform) and leaving all other settings unaltered. We observe that the number of digits obtained with the Baker transform in $d = 10$ can exceed even the number of digits obtained in $d = 8$ with the weight 3 Korobov transform.

Table 3
Comparison of timings using the QMC on CPUs & GPUs, the QMC CPUs only and Vegas as implemented in the Cuba library. The obtained relative accuracy refers to the finite real part of the integral including all prefactors mentioned in Section 4.3.

	QMC on GPUs		QMC on CPUs		Vegas	
	rel. acc.	Time (s)	rel. acc.	Time (s)	rel. acc.	Time (s)
`banana 3mass 3L`	$3.8 \cdot 10^{-11}$	15	$3.8 \cdot 10^{-11}$	23	$1.5 \cdot 10^{-3}$	39
`HZ nonplanar 2L`	$1.3 \cdot 10^{-3}$	24	$2.1 \cdot 10^{-3}$	28	$5.2 \cdot 10^{-3}$	27
`pentabox fin 2L`	$1.9 \cdot 10^{-4}$	42	$1.1 \cdot 10^{-3}$	133	$2.6 \cdot 10^{-3}$	139
`elliptic 2L`	$2.0 \cdot 10^{-6}$	9	$1.6 \cdot 10^{-6}$	33	$3.6 \cdot 10^{-4}$	104
`formfactor 4L`	$4.2 \cdot 10^{-7}$	258	$1.2 \cdot 10^{-5}$	235	$2.7 \cdot 10^{-4}$	986
`Nbox split b 2L`	$2.5 \cdot 10^{-3}$	60	$3.5 \cdot 10^{-2}$	77	$1.6 \cdot 10^{-1}$	177
`bubble 6L`	$8.5 \cdot 10^{-7}$	279	$1.1 \cdot 10^{-5}$	200	$5.7 \cdot 10^{-4}$	199

The source code of the program `1000_genz_demo`, used to perform the profiling presented in this section, is included in the `examples` folder of the stand-alone QMC distribution.

5.3. Timings for loop integrals

In Table 3, the timings for several of the examples described in Section 4 are compared using the QMC on CPUs & GPUs, the QMC on CPUs and Vegas as implemented in the Cuba library. The timings are performed on a machine with 2 x Intel Xeon Gold 6140 CPU @ 2.30 GHz CPUs (36 cores, 72 threads) and 4 x Nvidia Tesla V100 GPUs. The times reported in Table 3 correspond to the wall clock times for running the integration via the python interface of pySecDec. In particular, the numerical integration of *all* orders reported for the examples given in Section 4.3 is included in the timings. The integrands are summed before integration (`together=True`).

The timings given in Table 3 are obtained with the same parameters of the QMC as stated in Section 4.3 except for the number of samples (`minn`). The `maxeval` parameter is set to 1 such that the QMC does not iterate. Vegas is also run with a fixed number of function evaluations (`maxeval`) while the error goals `epsrel` and `epsabs` are set to 10^{-100} such that they do not trigger. The real and the imaginary part are integrated separately with Vegas (`real_complex_together=False`).

A special situation is encountered when integrating the 4-loop form factor with Vegas. The first output in verbose mode (`flags=2`) is printed to the screen only after about fifteen minutes. We suspect this long startup time is due to the rather large (879 MB) size of the dynamic pylink library in combination with the parallelization using `fork` as implemented in the Cuba integrators library.

It is generally faster to obtain many significant digits with the QMC integrator than with the Vegas integrator, especially when GPUs are available. For low-precision results however, Vegas can sometimes be faster.

6. Conclusions

We have presented a quasi-Monte Carlo integrator (QMC) which can be used both with GPUs and CPUs as a stand-alone library or within the pySecDec program. We have described the implementation of the QMC, based on a rank-1 shifted lattice rule, and given various examples of its usage. The examples of the use of the QMC within pySecDec comprise a 2-loop pentagon integral, integrals which are known to contain elliptic or hyperelliptic functions, a 4-loop form factor integral and a 6-loop 2-point function. The new version of pySecDec also contains other new

features, for example an improved algorithm to detect sector symmetries.

We have presented a novel approach to combine the QMC integration with importance sampling. We have investigated how the $\mathcal{O}(1/n)$ scaling of the error estimate depends on the dimension and form of the integrand, in particular on the transformation used to achieve a periodic integrand. In agreement with Refs. [40,41], we have demonstrated that rank-1 shifted lattice rules can considerably outperform integrators based on the Monte Carlo method. We also confirm that, in many cases, the use of GPUs (rather than CPUs) can lead to a speed-up of an order of magnitude or more. This implies that the number of accurate digits which can be computed in a reasonable amount of time using our implementation is often beyond that which can be reached using Vegas-like Monte Carlo integration. It should be noted, however, that the functions produced by sector decomposition are typically continuous and smooth enough to achieve $\mathcal{O}(1/n)$ scaling, while this is not necessarily the case for other integrands, as they occur for example in NNLO phase space integrals based on analytic subtraction of doubly unresolved real radiation.

We believe that the method presented here, along with the easy-to-use, publicly available implementation, can boost the numerical evaluation of multi-loop amplitudes with several mass scales to an unprecedented level of automation, speed and accuracy.

The stand-alone version of the QMC integrator is publicly available at https://github.com/mppmu/qmc. The new version of pySecDec is available at https://github.com/mppmu/secdec/releases and the online documentation can be found at https://secdec.readthedocs.io.

Acknowledgements

We would like to thank Tom Zirke for collaboration on previous versions of the code and Oliver Schulz for providing us access and support concerning the GPU usage. This research was supported in part by the COST Action CA16201 ('Particleface') of the European Union, and by the Swiss National Science Foundation (SNF), Switzerland under grant number 200020-175595. SB gratefully acknowledges financial support by the ERC Starting Grant "MathAm" (39568). The research of JS was supported by the European Union through the ERC Advanced Grant MC@NNLO (340983).

Appendix. API documentation

The QMC class has 7 template parameters:

- `T` the return type of the function to be integrated (assumed to be a real or complex floating point type)
- `D` the argument type of the function to be integrated (assumed to be a floating point type)
- `M` the maximum number of integration variables of any integrand that will be passed to the integrator
- `P` an integral transform to be applied to the integrand before integration
- `F` a function to be fitted to the inverse cumulative distribution function of the integrand in each dimension, used to reduce the variance of the integrand (default: `fitfunctions::None::template type`)
- `G` a c++11 style pseudo-random number engine (default: `std::mt19937_64`)
- `H` a c++11 style uniform real distribution (default: `std::uniform_real_distribution<D>`)

Internally, unsigned integers are assumed to be of type `U = unsigned long long int`.

Typically the return type `T` and argument type `D` are set to type double (for real numbers), `std::complex<double>` (for complex numbers on the CPU only) or `thrust::complex<double>` (for complex numbers on the GPU and CPU). In principle, the QMC library supports integrating other floating point types (e.g. quadruple precision, arbitrary precision, etc.), though they must be compatible with the relevant STL library functions or provide compatible overloads.

To integrate alternative floating point types, first include the header(s) defining the new type into your project and set the template arguments of the class `T` and `D` to your type. The following standard library functions must be compatible with your type or a compatible overload must be provided:

- `sqrt`, `abs`, `modf`, `pow`
- `std::max`, `std::min`

If your type is not intended to represent a real or complex type number then you may also need to overload functions required for calculating the error resulting from the numerical integration, see the files `src/overloads/real.hpp` and `src/overloads/complex.hpp`.

Example `9_boost_minimal_demo` demonstrates how to instantiate the QMC with a non-standard type (`boost::multiprecision::cpp_bin_float_quad`). To compile this example you will need the `boost` library available on your system.

A.1. Public fields

Logger logger A wrapped `std::ostream` object to which log output from the library is written.

To write the text output of the library to a particular file, first `#include <fstream>`, create a `std::ofstream` instance pointing to your file then set the logger of the integrator to the `std::ofstream`. For example to output very detailed output to the file `myoutput.log`:

```
1 std::ofstream out_file("myoutput.log");
2 integrators::Qmc<double,double,MAXVAR,integrators::
      transforms::Korobov<3>::type> integrator;
3 integrator.verbosity=3;
4 integrator.logger = out_file;
```

Default: `std::cout`.

G randomgenerator A c++11 style pseudo-random number engine.

The seed of the pseudo-random number engine can be changed via the seed member function of the pseudo-random number engine. For total reproducibility you may also want to set `cputhreads = 1` and `devices = {-1}` which disables multi-threading, this helps to ensure that the floating point operations are done in the same order each time the code is run. For example:

```
1 integrators::Qmc<double,double,MAXVAR,integrators::
      transforms::Korobov<3>::type> integrator;
2 integrator.randomgenerator.seed(1) // seed = 1
3 integrator.cputhreads = 1; // no multi-threading
4 integrator.devices = {-1}; // cpu only
```

Default: `std::mt19937_64` seeded with a call to `std::random_device`.

U minn The minimum lattice size that should be used for integration. If a lattice of the requested size is not available then `n` will be the size of the next available lattice with at least `minn` points.

Default: `8191`.

U minm The minimum number of random shifts of the lattice `m` that should be used to estimate the error of the result. Typically 10 to 50.

Default: `32`.

D epsrel The relative error that the QMC should attempt to achieve.

Default: `0.01`.

D epsabs The absolute error that the QMC should attempt to achieve. For real types the integrator tries to find an estimate `E` for the integral `I` which fulfils `|E-I| <= max (epsabs, epsrel*I)`. For complex types the goal is controlled by the `errormode` setting.

Default: `1e-7`.

U maxeval The (approximate) maximum number of function evaluations that should be performed while integrating. The actual number of function evaluations can be slightly larger if there is not a suitably sized lattice available.

Default: `1000000`.

U maxnperpackage Maximum number of points to compute per thread per work package.

Default: `1`.

U maxmperpackage Maximum number of shifts to compute per thread per work package.

Default: `1024`.

ErrorMode errormode Controls the error goal that the library attempts to achieve when the integrand return type is a complex type. For real types the `errormode` setting is ignored. Possible values:

- `all` — try to find an estimate `E` for the integral `I` which fulfils `|E-I| <= max(epsabs, epsrel*I)` for each component (real and imaginary) separately,
- `largest` — try to find an estimate `E` for the integral `I` such that $\max(|Re[E] - Re[I]|, |Im[E] - Im[I]|) \leq \max(\epsilon_{\text{abs}}, \epsilon_{\text{rel}} \cdot \max(|Re[I]|, |Im[I]|))$, i.e. to achieve either the `epsabs` error goal or that the largest error is smaller than `epsrel` times the value of the largest component (either real or imaginary).

Default: `all`.

U cputhreads The number of CPU threads that should be used to evaluate the integrand function. If GPUs are used 1 additional CPU thread per device will be launched for communicating with the device.

Default: `std::thread::hardware_concurrency()`.

U cudablocks The number of blocks to be launched on each CUDA device.

Default: (determined at run time).

U cudathreadsperblock The number of threads per block to be launched on each CUDA device. CUDA kernels launched by the QMC library have the execution configuration `<<< cudablocks, cudathreadsperblock >>>`. For more information on how to optimally configure these parameters for your hardware and/or integral refer to the Nvidia guidelines.

Default: (determined at run time).

`std::set<int> devices` A set of devices on which the integrand function should be evaluated. The device id `-1` represents all CPUs present on the system, the field `cputhreads` can be used to control the number of CPU threads spawned. The indices `0,1, . . .` are device ids of CUDA devices present on the system.

Default: `-1,0,1, . . . ,nd` where `nd` is the number of CUDA devices detected on the system.

`std::map<U,std::vector<U>> generatingvectors` A map of available generating vectors which can be used to generate a lattice. The implemented QMC algorithm requires that the generating vectors be generated with a prime lattice size. By default the library uses generating vectors with 100 components, thus it supports integration of functions with up to 100 dimensions. The default generating vectors have been generated with lattice size chosen as the next prime number above $(110/100)^i \cdot 1020$ for `i` between `0` and `152`, additionally the lattice $2^{31} - 1$ (`INT_MAX` for `int32`) is included.

Default: `cbcpt_dn1_100()`.

U evaluateminn The minimum lattice size that should be used by the `evaluate` function to evaluate the integrand, if variance reduction is enabled these points are used for fitting the inverse cumulative distribution function. If a lattice of the requested size is not available then n will be the size of the next available lattice with at least `evaluateminn` points.

Default: `100000`.

U verbosity Possible values: `0,1,2,3`. Controls the verbosity of the output to `logger` of the QMC library.

0 — no output,
1 — key status updates and statistics,
2 — detailed output, useful for debugging,
3 — very detailed output, useful for debugging.

Default: `0`.

size_t fitstepsize Controls the number of points included in the fit used for variance reduction. A step size of `x` includes (after sorting by value) every `x`th point in the fit.

Default: `10`.

size_t fitmaxiter See `maxiter` in the non-linear least-squares fitting `GSL` documentation.

Default: `40`.

double fitxtol See `xtol` in the non-linear least-squares fitting `GSL` documentation.

Default: `3e-3`.

double fitgtol See `gtol` in the non-linear least-squares fitting `GSL` documentation.

Default: `1e-8`.

double fitftol See `ftol` in the non-linear least-squares fitting GSL documentation.

Default: `1e-8`.

gsl_multifit_nlinear_parameters fitparametersgsl See `gsl_multifit_nlinear_parameters` in the non-linear least-squares fitting GSL documentation.

Default: `{}`.

A.2. Public member functions

U get_next_n(U preferred_n) Returns the lattice size n of the lattice in `generatingvectors` that is greater than or equal to `preferred_n`. This represents the size of the lattice that would be used for integration if `minn` was set to `preferred_n`.

`template <typename I> result<T,U> integrate(I& func)` Integrates the function `func` in d dimensions using the integral transform `transform`. The result is returned in a `result` struct with the following members:

- `integral` — the result of the integral
- `error` — the estimated absolute error of the result
- `n` — the size of the largest lattice used during integration
- `m` — the number of shifts of the largest lattice used during integration.
- `U iterations` — the number of iterations used during integration
- `U evaluations` — the total number of function evaluations during integration

The functor `func` must define its dimension as a public member variable `number_of_integration_variables`.

Calls: `get_next_n`.

`template <typename I> samples<T,D> evaluate(I& func)` Evaluates the functor `func` on a lattice of size greater than or equal to `evaluateminn`. The samples are returned in a `samples` struct with the following members:

- `std::vector<U> z` — the generating vector of the lattice used to produce the samples
- `std::vector<D> d` — the random shift vector used to produce the samples
- `std::vector<T> r` — the values of the integrand at each randomly shifted lattice point
- `U n` — the size of the lattice used to produce the samples
- `D get_x(const U sample_index, const U integration_variable_index)` — a function which returns the argument (specified by `integration_variable_index`) used to evaluate the integrand for a specific sample (specified by `sample_index`).

The functor `func` must define its dimension as a public member variable `number_of_integration_variables`.

Calls: `get_next_n`.

`template <typename I> typename F<I,D,M>::transform_t fit(I& func)` Fits a function (specified by the type `F` of the integrator) to the inverse cumulative distribution function of the integrand dimension-by-dimension and returns a functor representing the new integrand after this variance reduction procedure.

The functor `func` must define its dimension as a public member variable `number_of_integration_variables`.

Calls: `get_next_n`, `evaluate`.

Table A.1
Types of generating vectors distributed with the program.

Name	Max. dimension	Computed via	Lattice sizes
`cbcpt_dn1_100`	100	*fastrank1pt.m* tool [63]	1021 – 2147483647
`cbcpt_dn2_6`	6	*fastrank1pt.m* tool [63]	65521 – 2499623531
`cbcpt_cfftw1_6`	6	CBC tool based on [64]	2500000001 – 15173222401

Table A.2
Types of periodizing transformations distributed with the program.

Name	Description
`Korobov<r_0,r_1>`	A polynomial integral transform with weight $\propto x^{r_0}(1-x)^{r_1}$
`Korobov<r>`	A polynomial integral transform with weight $\propto x^{r}(1-x)^{r}$
`Sidi<r>`	A trigonometric integral transform with weight $\propto \sin^{r}(\pi x)$
`Baker`	The baker's transformation, $\phi(x) = 1 - \lvert 2x-1\rvert$
`None`	The trivial transform, $\phi(x) = x$

A.3. Generating vectors

We offer generating vectors for different lattice sizes n, and also for different maximal dimensions s: $s \leq 6$ or $s \leq 100$. The generating vectors which are distributed with the version described in this paper are summarized in Table A.1. We used the so-called Component-By-Component (CBC) construction [47], computed using partly D. Nuyens' `fastrank1pt.m` tool [63] and, for very large lattice sizes, our own CBC tool based on the FFTW algorithm [64].

The generating vectors distributed with the code are produced for Korobov spaces with smoothness $\alpha = 2$, in the notation of Ref. [65] we use:

- Kernel $\omega(x) = 2\pi^2(x^2 - x + 1/6)$,
- Weights $\gamma_i = 1/s$ for $i = 1, \ldots, s$,
- Parameters $\beta_i = 1$ for $i = 1, \ldots, s$.

The generating vectors used by the QMC can be selected by setting the integrator's `generatingvectors` member variable. Example (assuming an integrator instance named `integrator`):

```
integrator.generatingvectors = integrators::generatingvectors::
    cbcpt_dn2_6();
```

If you prefer to use custom generating vectors and/or 100 dimensions and/or 15173222401 lattice points are not enough, you can supply your own generating vectors. Compute your generating vectors using another tool then put them into a map and set `generatingvectors`. For example, to instruct the QMC to use only two generating vectors ($\mathbf{z} = (1, 3)$ for $n = 7$ and $\mathbf{z} = (1, 7)$ for $n = 11$) the `generatingvectors` map would be set as follows:

```
1 std::map<unsigned long long int,std::vector<unsigned long long
      int>> my_generating_vectors = { {7, {1,3}}, {11, {1,7}} };
2 integrators::Qmc<double,double,10> integrator;
3 integrator.generatingvectors = my_generating_vectors;
```

A.4. Integral transforms

The integral transforms distributed with the QMC are listed in Table A.2. The integral transform used by the QMC can be selected when constructing the QMC. Example (assuming a real type integrator instance named integrator):

Table A.3
Types of fit functions distributed with the program.

Name	Description
`PolySingular`	A 3rd order polynomial with two additional $1/(p-x)$ terms, $f(x) = \frac{\lvert p_2\rvert(x(p_0-1))}{(p_0-x)} + \frac{\lvert p_3\rvert(x(p_1-1))}{(p_1-x)} + x(p_4 + x(p_5 + x(1 - \lvert p_2\rvert - \lvert p_3\rvert - p_4 - p_5)))$
`None`	The trivial transform, $f(x) = x$

```
integrators::Qmc<double,double,10,integrators::transforms::
    Korobov<5,3>::type> integrator;
```

instantiates an integrator which applies a weight ($r_0 = 5, r_1 = 3$) Korobov transform to the integrand before integration.

A.5. Fit functions

The fit function used by the QMC can be selected when constructing the QMC. These functions are used to approximate the inverse cumulative distribution function of the integrand dimension-by-dimension. Example (assuming a real type integrator instance named `integrator`):

```
integrators::Qmc<double,double,10,integrators::transforms::
    Korobov<3>::type,integrators::fitfunctions::PolySingular::
    type> integrator;
```

instantiates an integrator which reduces the variance of the integrand by fitting a `PolySingular` type function before integration. Possible fit functions are shown in Table A.3.

References

[1] S. Laporta, E. Remiddi, Nuclear Phys. B704 (2005) 349–386, http://dx.doi.org/10.1016/j.nuclphysb.2004.10.044, arXiv:hep-ph/0406160.
[2] L. Adams, E. Chaubey, S. Weinzierl, Phys. Rev. Lett. 118 (14) (2017) 141602, http://dx.doi.org/10.1103/PhysRevLett.118.141602, arXiv:1702.04279.
[3] S. Abreu, R. Britto, C. Duhr, E. Gardi, Phys. Rev. Lett. 119 (5) (2017) 051601, http://dx.doi.org/10.1103/PhysRevLett.119.051601, arXiv:1703.05064.
[4] A. Primo, L. Tancredi, Nuclear Phys. B921 (2017) 316–356, http://dx.doi.org/10.1016/j.nuclphysb.2017.05.018, arXiv:1704.05465.
[5] J.L. Bourjaily, A.J. McLeod, M. Spradlin, M. von Hippel, M. Wilhelm, Phys. Rev. Lett. 120 (12) (2018) 121603, http://dx.doi.org/10.1103/PhysRevLett.120.121603, arXiv:1712.02785.
[6] J. Broedel, C. Duhr, F. Dulat, L. Tancredi, J. High Energy Phys. 05 (2018) 093, http://dx.doi.org/10.1007/JHEP05(2018)093, arXiv:1712.07089.
[7] L. Adams, S. Weinzierl, Phys. Lett. B781 (2018) 270–278, http://dx.doi.org/10.1016/j.physletb.2018.04.002, arXiv:1802.05020.
[8] J. Broedel, C. Duhr, F. Dulat, B. Penante, L. Tancredi, J. High Energy Phys. 08 (2018) 014, http://dx.doi.org/10.1007/JHEP08(2018)014, arXiv:1803.10256.
[9] R.N. Lee, A.V. Smirnov, V.A. Smirnov, J. High Energy Phys. 07 (2018) 102, http://dx.doi.org/10.1007/JHEP07(2018)102, arXiv:1805.00227.
[10] J.L. Bourjaily, Y.-H. He, A.J. Mcleod, M. Von Hippel, M. Wilhelm, Phys. Rev. Lett. 121 (7) (2018) 071603, http://dx.doi.org/10.1103/PhysRevLett.121.071603, arXiv:1805.09326.
[11] J. Blümlein, C. Schneider, Internat. J. Modern Phys. A33 (17) (2018) 1830015, http://dx.doi.org/10.1142/S0217751X18300156, arXiv:1809.02889.
[12] J. Broedel, C. Duhr, F. Dulat, B. Penante, L. Tancredi, Elliptic Feynman Integrals and Pure Functions, 2018, arXiv:1809.10698.
[13] J.L. Bourjaily, A.J. McLeod, M. von Hippel, M. Wilhelm, A (Bounded) Bestiary of Feynman Integral Calabi-Yau Geometries, 2018, arXiv:1810.07689.
[14] K. Hepp, Comm. Math. Phys. 2 (1966) 301–326.
[15] M. Roth, A. Denner, Nuclear Phys. B479 (1996) 495–514, arXiv:hep-ph/9605420.
[16] T. Binoth, G. Heinrich, Nuclear Phys. B585 (2000) 741–759, arXiv:hep-ph/0004013.
[17] G. Heinrich, Internat. J. Modern Phys. A23 (2008) 1457–1486, http://dx.doi.org/10.1142/S0217751X08040263, arXiv:0803.4177.
[18] S. Becker, S. Weinzierl, Eur. Phys. J. C73 (2) (2013) 2321, http://dx.doi.org/10.1140/epjc/s10052-013-2321-1, arXiv:1211.0509.
[19] G.F.R. Sborlini, F. Driencourt-Mangin, G. Rodrigo, J. High Energy Phys. 10 (2016) 162, http://dx.doi.org/10.1007/JHEP10(2016)162, arXiv:1608.01584.
[20] A. Freitas, Prog. Part. Nucl. Phys. 90 (2016) 201–240, http://dx.doi.org/10.1016/j.ppnp.2016.06.004, arXiv:1604.00406.
[21] E. de Doncker, F. Yuasa, K. Kato, T. Ishikawa, J. Kapenga, O. Olagbemi, Comput. Phys. Comm. 224 (2018) 164–185, http://dx.doi.org/10.1016/j.cpc.2017.11.001, arXiv:1702.04904.
[22] J. Gluza, T. Jelinski, D.A. Kosower, Phys. Rev. D95 (7) (2017) 076016, http://dx.doi.org/10.1103/PhysRevD.95.076016, arXiv:1609.09111.
[23] J. Usovitsch, I. Dubovyk, T. Riemann, PoS LL2018 (2018) 046, arXiv:1810.04580.
[24] J. Baglio, F. Campanario, S. Glaus, M. Mühlleitner, M. Spira, J. Streicher, Gluon Fusion into Higgs Pairs at NLO QCD and the Top Mass Scheme, 2018, arXiv:1811.05692.
[25] J.R. Andersen, et al., Les Houches 2017: Physics at TeV Colliders Standard Model Working Group Report, 2018. arXiv:1803.07977.
[26] A. Blondel, et al., Mini Workshop on Precision EW and QCD Calculations for the FCC Studies: Methods and Techniques; CERN, Geneva, Switzerland, January 12-13, 2018, 2018.
[27] C. Bogner, S. Weinzierl, Comput. Phys. Comm. 178 (2008) 596–610, http://dx.doi.org/10.1016/j.cpc.2007.11.012, arXiv:0709.4092.
[28] J. Gluza, K. Kajda, T. Riemann, V. Yundin, Eur. Phys. J. C71 (2011) 1516, http://dx.doi.org/10.1140/epjc/s10052-010-1516-y, arXiv:1010.1667.
[29] T. Ueda, J. Fujimoto, PoS ACAT08 (2008) 120, arXiv:0902.2656.
[30] A. Smirnov, M. Tentyukov, Comput. Phys. Comm. 180 (2009) 735–746, http://dx.doi.org/10.1016/j.cpc.2008.11.006, arXiv:0807.4129.
[31] A. Smirnov, V. Smirnov, M. Tentyukov, Comput. Phys. Comm. 182 (2011) 790–803, http://dx.doi.org/10.1016/j.cpc.2010.11.025, arXiv:0912.0158.
[32] A.V. Smirnov, Comput. Phys. Comm. 185 (2014) 2090–2100, http://dx.doi.org/10.1016/j.cpc.2014.03.015, arXiv:1312.3186.
[33] A.V. Smirnov, Comput. Phys. Comm. 204 (2016) 189–199, http://dx.doi.org/10.1016/j.cpc.2016.03.013, arXiv:1511.03614.
[34] S. Borowka, G. Heinrich, S.P. Jones, M. Kerner, J. Schlenk, T. Zirke, Comput. Phys. Comm. 196 (2015) 470–491, http://dx.doi.org/10.1016/j.cpc.2015.05.022, arXiv:1502.06595.
[35] S. Borowka, G. Heinrich, S. Jahn, S.P. Jones, M. Kerner, J. Schlenk, T. Zirke, Comput. Phys. Comm. 222 (2018) 313–326, http://dx.doi.org/10.1016/j.cpc.2017.09.015, arXiv:1703.09692.
[36] S. Borowka, T. Gehrmann, D. Hulme, J. High Energy Phys. 08 (2018) 111, http://dx.doi.org/10.1007/JHEP08(2018)111, arXiv:1804.06824.
[37] T. Hahn, Comput. Phys. Comm. 168 (2005) 78–95, http://dx.doi.org/10.1016/j.cpc.2005.01.010, arXiv:hep-ph/0404043.
[38] T. Hahn, Concurrent Cuba, 2014, arXiv:1408.6373.
[39] J. Dick, F.Y. Kuo, I.H. Sloan, Acta Numer. 22 (2013) 133–288.
[40] Z. Li, J. Wang, Q.-S. Yan, X. Zhao, Chin. Phys. C 40, No. 3 (2016) 033103, http://dx.doi.org/10.1088/1674-1137/40/3/033103, arXiv:1508.02512.
[41] E. de Doncker, A. Almulihi, F. Yuasa, J. Phys. Conf. Ser. 1085 (5) (2018) 052005, http://dx.doi.org/10.1088/1742-6596/1085/5/052005.
[42] S. Borowka, N. Greiner, G. Heinrich, S. Jones, M. Kerner, J. Schlenk, U. Schubert, T. Zirke, Phys. Rev. Lett. 117 (1) (2016) 012001; Erratum:, Phys. Rev. Lett. 117 (7) (2016) 079901, http://dx.doi.org/10.1103/PhysRevLett.117.079901, arXiv:1604.06447 http://dx.doi.org/10.1103/PhysRevLett.117.012001.
[43] S. Borowka, N. Greiner, G. Heinrich, S.P. Jones, M. Kerner, J. Schlenk, T. Zirke, J. High Energy Phys. 10 (2016) 107, http://dx.doi.org/10.1007/JHEP10(2016)107, arXiv:1608.04798.
[44] S.P. Jones, M. Kerner, G. Luisoni, Phys. Rev. Lett. 120 (16) (2018) 162001, http://dx.doi.org/10.1103/PhysRevLett.120.162001, arXiv:1802.00349.
[45] F.Y. Kuo, D. Nuyens, Lecture Notes: A Practical Guide to Quasi-Monte Carlo Methods, National Chiao Tung University & National Taiwan University, 2016.
[46] I.H. Sloan, H. Woniakowski, J. Complexity 14 (1) (1998) 1–33, http://dx.doi.org/10.1006/jcom.1997.0463, URL http://www.sciencedirect.com/science/article/pii/S0885064X97904635.
[47] D. Nuyens, R. Cools, Math. Comp. 75 (254) (2006) 903–920.
[48] N.M. Korobov, Number-Theoretic Methods in Approximate Analysis, Fizmatgiz, Moscow, 1963.
[49] D.P. Laurie, J. Comput. Appl. Math. 66 (1) (1996) 337–344, http://dx.doi.org/10.1016/0377-0427(95)00196-4, Proceedings of the Sixth International Congress on Computational and Applied Mathematics. URL http://www.sciencedirect.com/science/article/pii/0377042795001964.
[50] F.Y. Kuo, I.H. Sloan, H. Woźniakowski, Numer. Algorithms 46 (4) (2007) 369–391, http://dx.doi.org/10.1007/s11075-007-9145-8.
[51] A. Sidi, A New Variable Transformation for Numerical Integration, Birkhäuser Basel, Basel, 1993, pp. 359–373, URL https://doi.org/10.1007/978-3-0348-6338-4_27.
[52] F.J. Hickernell, in: K.T. Fang, F.J. Hickernell, H. Niederreiter (Eds.), Monte Carlo and Quasi-Monte Carlo Methods 2000, Springer, Berlin, 2002, p. 274289.
[53] G.P. Lepage, J. Comput. Phys. 27 (1978) 192, http://dx.doi.org/10.1016/0021-9991(78)90004-9.

[54] M. Galassi, et al., GNU Scientific Library Reference Manual - Third Edition, third ed., Network Theory Ltd, 2009.
[55] R. Bonciani, V. Del Duca, H. Frellesvig, J.M. Henn, F. Moriello, V.A. Smirnov, J. High Energy Phys. 12 (2016) 096, http://dx.doi.org/10.1007/JHEP12(2016)096, arXiv:1609.06685.
[56] A. Georgoudis, Y. Zhang, J. High Energy Phys. 12 (2015) 086, http://dx.doi.org/10.1007/JHEP12(2015)086, arXiv:1507.06310.
[57] A. von Manteuffel, E. Panzer, R.M. Schabinger, Phys. Rev. D93 (12) (2016) 125014, http://dx.doi.org/10.1103/PhysRevD.93.125014, arXiv:1510.06758.
[58] S. Jahn, PoS LL2018 (2018) 019, http://dx.doi.org/10.22323/1.303.0019.
[59] M.V. Kompaniets, E. Panzer, Phys. Rev. D96 (3) (2017) 036016, http://dx.doi.org/10.1103/PhysRevD.96.036016, arXiv:1705.06483.
[60] F.V. Tkachov, Phys. Lett. 100B (1981) 65–68, http://dx.doi.org/10.1016/0370-2693(81)90288-4.
[61] K.G. Chetyrkin, F.V. Tkachov, Nuclear Phys. B192 (1981) 159–204, http://dx.doi.org/10.1016/0550-3213(81)90199-1.
[62] A. Genz, in: P. Keast, G. Fairweather (Eds.), A Package for Testing Multiple Integration Subroutines, Springer Netherlands, Dordrecht, 1987, pp. 337–340, http://dx.doi.org/10.1007/978-94-009-3889-2_33.
[63] D. Nuyens, https://people.cs.kuleuven.be/~dirk.nuyens/fast-cbc/.
[64] M. Frigo, S. Johnson, et al., https://github.com/FFTW/fftw3.
[65] D. Nuyens, R. Cools, in: H. Niederreiter, D. Talay (Eds.), Monte Carlo and Quasi-Monte Carlo Methods 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 373–387.